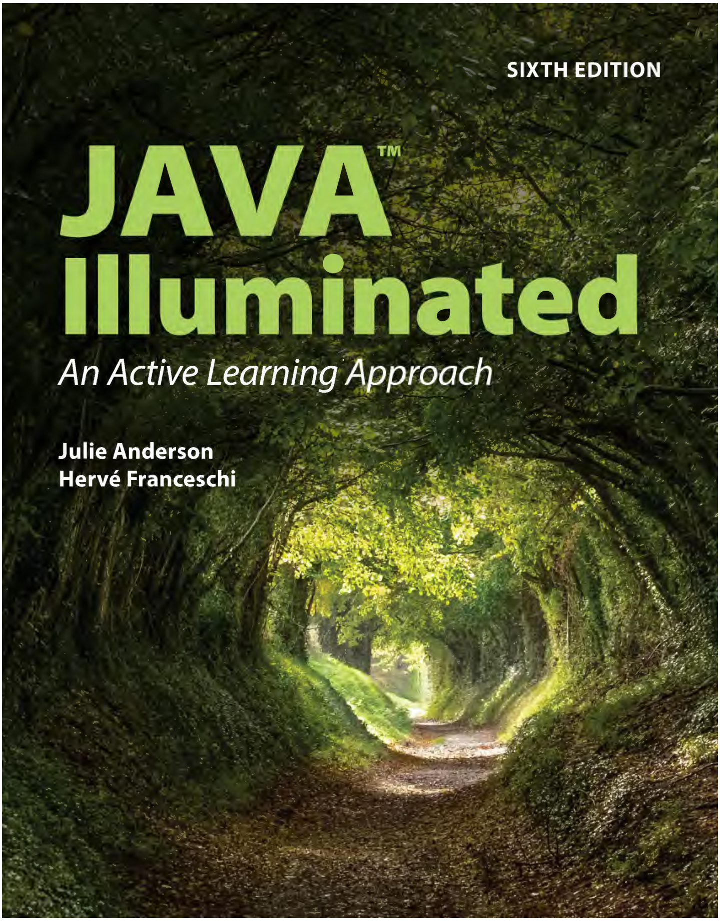
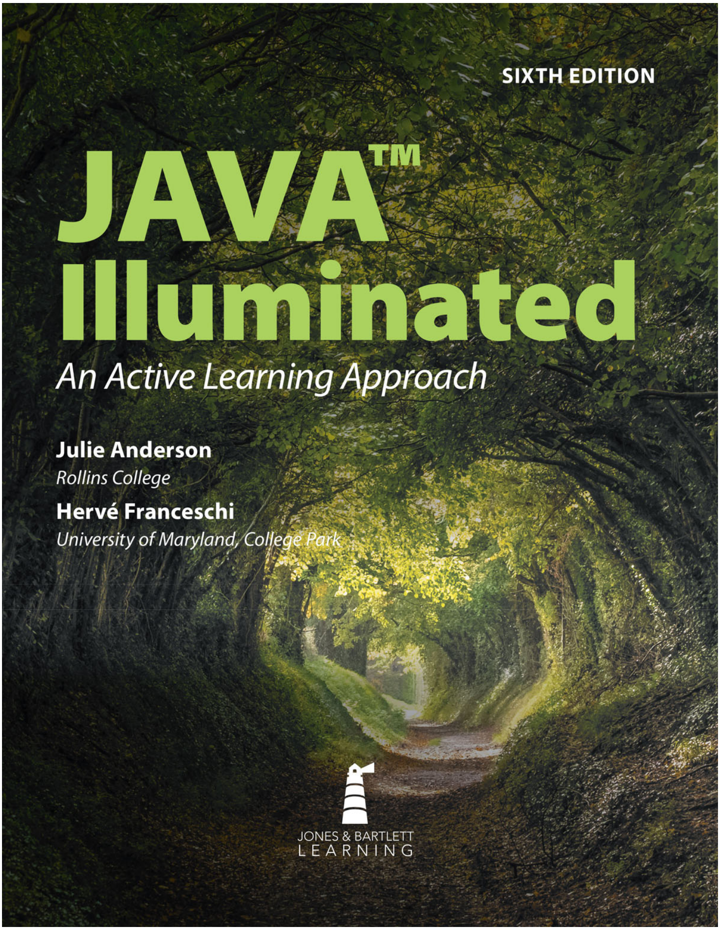


SIXTH EDITION

JAVA™ Illuminated

An Active Learning Approach

Julie Anderson
Hervé Franceschi



SIXTH EDITION

JAVA™ Illuminated

An Active Learning Approach

Julie Anderson
Rollins College
Hervé Franceschi
University of Maryland, College Park

JONES & BARTLETT
LEARNING

World Headquarters
Jones & Bartlett Learning
25 Mall Road
Burlington MA 01803
978-443-5000
info@jblearning.com
www.jblearning.com

Jones & Bartlett Learning books and products are available through most bookstores and online booksellers. To contact Jones & Bartlett Learning directly, call 800-832-0034, fax 978-443-8000, or visit our website, www.jblearning.com.

Substantial discounts on bulk quantities of Jones & Bartlett Learning publications are available to corporations, professional associations, and other qualified organizations. For details and specific discount information, contact the special sales department at Jones & Bartlett Learning via the above contact information or send an email to specialsales@jblearning.com.

Copyright © 2024 by Jones & Bartlett Learning, LLC, an Ascend Learning Company

All rights reserved. No part of the material protected by this copyright may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

The content, statements, views, and opinions herein are the sole expression of the respective authors and not that of Jones & Bartlett Learning, LLC. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not constitute or imply its endorsement or recommendation by Jones & Bartlett Learning, LLC and such reference shall not be used for advertising or product endorsement purposes. All trademarks displayed are the trademarks of the parties noted herein. *Java Illuminated: An Active*

Learning Approach, Sixth Edition is an independent publication and has not been authorized, sponsored, or otherwise approved by the owners of the trademarks or service marks referenced in this product.

There may be images in this book that feature models; these models do not necessarily endorse, represent, or participate in the activities represented in the images. Any screenshots in this product are for educational and instructive purposes only. Any individuals and scenarios featured in the case studies throughout this product may be real or fictitious but are used for instructional purposes only.

25141-8

Production Credits
Vice President, Product Management: Marisa R. Urbano
Vice President, Content Strategy and Implementation: Christine Emerton
Director, Product Management: Ray Chew
Product Manager: Edward Hinman
Director, Content Management: Donna Gridley
Manager, Content Strategy: Carolyn Pershouse
Content Strategist: Melissa Duffy
Content Coordinator: Mark Restuccia
Director, Project Management and Content Services: Karen Scott
Manager, Project Management: Jackie Reynen
Project Manager: Jennifer Risden
Senior Digital Project Specialist: Angela Dooley
Director, Marketing: Andrea DeFronzo
Content Services Manager: Colleen Lamy
VP, Manufacturing and Inventory Control: Therese Connell
Product Fulfillment Manager: Wendy Kilborn
Composition: Straive
Cover Design: Briana Yates
Media Development Editor: Faith Brosnan
Rights & Permissions Manager: John Rusk
Rights Specialist: James Fortney
Cover Image (Title Page, Part Opener, Chapter Opener): © Lois GoBe/ Shutterstock.
Printing and Binding: LSC Communications

Library of Congress Cataloging-in-Publication Data
Names: Anderson, Julie A., author. | Franceschi, Hervé, author.
Title: Java(TM) illuminated: an active learning approach / Julie A. Anderson, MS Rollins College Hervé J. Franceschi Loyola University Maryland.
Description: Sixth edition. | Burlington, Massachusetts : Jones & Bartlett Learning, [2024] | Includes index.
Identifiers: LCCN 2022015401 | ISBN 9781284250480 (paperback)
Subjects: LCSH: Java (Computer program language)
Classification: LCC QA76.73.J3 A533 2022 | DDC 005.13/3--dc23/

eng/20220527
LC record available at https://lccn.loc.gov/2022015401
6048

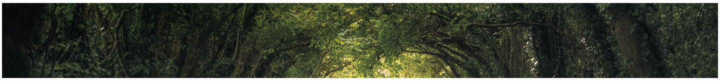
Printed in the United States of America
26 25 24 23 22   10 9 8 7 6 5 4 3 2 1

To the memory of my parents,
Glenn and Rosemary Austin, my first teachers.

– Julie Anderson

A ma mère, trop tôt disparue, et à mon père.

– Hervé Franceschi



© Lois GoBe/Shutterstock.

Contents

Preface
Acknowledgments

Chapter 1 Introduction to Programming and the Java Language
- Introduction
- 1.1 Basic Computer Concepts
  - 1.1.1 Hardware
  - 1.1.2 Operating Systems
  - 1.1.3 Application Software
  - 1.1.4 Computer Networks and the Internet
    - *Computer Networks*
    - *The Internet*
- 1.2 Practice Activity: Displaying System Configuration
  - 1.2.1 Displaying Windows Configuration Information
  - 1.2.2 Displaying Mac OS Configuration Information
- 1.3 Data Representation
  - 1.3.1 Binary Numbers
  - 1.3.2 Using Hexadecimal Numbers to Represent Binary Numbers
  - 1.3.3 Representing Characters with the Unicode Character Set
- 1.4 Programming Languages
  - 1.4.1 High- and Low-Level Languages
  - 1.4.2 An Introduction to Object-Oriented Programming
  - 1.4.3 The Java Language
- 1.5 An Introduction to Programming

- 1.5.1 Programming Basics
- 1.5.2 Program Design with Pseudocode
- 1.5.3 Developing a Java Application
- 1.5.4 Programming Activity 1: Writing a First Java Application
  - *Debugging Techniques*
  - *Testing Techniques*
- 1.5.5 Making a JAR File
- 1.6 Chapter Summary
- 1.7 Exercises, Problems, and Projects
  - 1.7.1 Multiple Choice Exercises
  - 1.7.2 Converting Numbers
  - 1.7.3 General Questions
  - 1.7.4 Technical Writing
  - 1.7.5 Group Project (for a group of 1, 2, or 3 students)

Chapter 2 Programming Building Blocks— Java Basics
- Introduction
- 2.1 Java Application Structure
- 2.2 Data Types, Variables, and Constants
  - 2.2.1 Declaring Variables
  - 2.2.2 Integer Data Types
  - 2.2.3 Floating-Point Data Types
  - 2.2.4 Character Data Type
  - 2.2.5 Boolean Data Type
  - 2.2.6 The Assignment Operator, Initial Values, and Literals
  - 2.2.7 *String* Literals and Escape Sequences
  - 2.2.8 Text Blocks
  - 2.2.9 Constants
- 2.3 Expressions and Arithmetic Operators
  - 2.3.1 The Assignment Operator and Expressions
  - 2.3.2 Arithmetic Operators
  - 2.3.3 Operator Precedence
  - 2.3.4 Programming Activity 1: Converting Inches to Centimeters
  - 2.3.5 Integer Division and Modulus

- 2.3.6 Division by Zero
- 2.3.7 Mixed-Type Arithmetic and Type Casting
- 2.3.8 Shortcut Operators

2.4 Programming Activity 2: Temperature Conversion

2.5 Chapter Summary

2.6 Exercises, Problems, and Projects

- 2.6.1 Multiple Choice Exercises
- 2.6.2 Reading and Understanding Code
- 2.6.3 Fill In the Code
- 2.6.4 Identifying Errors in Code
- 2.6.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
- 2.6.6 Write a Short Program
- 2.6.7 Programming Projects
- 2.6.8 Technical Writing

Chapter 3 Object-Oriented Programming, Part 1: Using Classes

Introduction

3.1 Class Basics and Benefits

3.2 Creating Objects Using Constructors

3.3 Calling Methods

3.4 Using Object References

3.5 More on the *String* Class

- *The length Method*
- *The charAt Method*
- *The indexOf Methods*
- *The substring Methods*
- *The isBlank and strip Methods*
- *String Processing*

3.6 Programming Activity 1: Calling Methods

3.7 The Java Class Library

3.8 Formatting Output with the *DecimalFormat* Class

3.9 Generating Random Numbers with the *Random* Class

3.10 Input from the Console Using the *Scanner* Class

3.11 Calling *Static* Methods and Using *Static* Class Variables

3.12 Using *System.in* and *System.out*

- 3.13 The *Math* Class
    - *The pow Method*
    - *The round Method*
    - *The min and max Methods*
- 3.14 Formatting Output with the *NumberFormat* Class
- 3.15 The *Integer, Double, Character,* and Other Wrapper Classes
- 3.16 Programming Activity 2: Using Predefined Classes
- 3.17 Chapter Summary
- 3.18 Exercises, Problems, and Projects
    - 3.18.1 Multiple Choice Exercises
    - 3.18.2 Reading and Understanding Code
    - 3.18.3 Fill In the Code
    - 3.18.4 Identifying Errors in Code
    - 3.18.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
    - 3.18.6 Write a Short Program
    - 3.18.7 Programming Projects
    - 3.18.8 Technical Writing
    - 3.18.9 Group Project (for a group of 1, 2, or 3 students)

Chapter 4 Introduction to Graphical Applications

- Introduction
- 4.1 JavaFX Application Structure
- 4.2 The Graphics Coordinate System and Color
- 4.3 Drawing Shapes and Text
- 4.4 Drawing Custom Shapes
- 4.5 Programming Activity 1: Writing an Application with Graphics
- 4.6 Chapter Summary
- 4.7 Exercises, Problems, and Projects
    - 4.7.1 Multiple Choice Exercises
    - 4.7.2 Reading and Understanding Code
    - 4.7.3 Fill In the Code
    - 4.7.4 Identifying Errors in Code
    - 4.7.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
    - 4.7.6 Write a Short Program

- 4.7.7 Programming Projects
- 4.7.8 Technical Writing
- 4.7.9 Group Project (for a group of 1, 2, or 3 students)

Chapter 5 Flow of Control, Part 1: Selection

- Introduction
- 5.1 Forming Conditions
  - 5.1.1 Equality Operators
  - 5.1.2 Relational Operators
  - 5.1.3 Logical Operators
    - *DeMorgan's Laws*
- 5.2 Simple Selection with *if*
- 5.3 Selection Using *if/else*
    - *Block Scope*
- 5.4 Selection Using *if/else if*
- 5.5 Sequential and Nested *if/else* Statements
  - 5.5.1 Sequential *if/else* Statements
    - *Finding the Minimum or Maximum Values*
  - 5.5.2 Nested *if/else* Statements
    - *Dangling else*
- 5.6 Testing Techniques for *if/else* Statements
- 5.7 Programming Activity 1: Working with *if/else*
- 5.8 Comparing Floating-Point Numbers
- 5.9 Comparing Objects
  - 5.9.1 The *equals* Method
  - 5.9.2 *String* Comparison Methods
- 5.10 The Conditional Operator (?:)
- 5.11 The *switch* Statement
- 5.12 Programming Activity 2: Using the *switch* Statement
- 5.13 Chapter Summary
- 5.14 Exercises, Problems, and Projects
  - 5.14.1 Multiple Choice Exercises
  - 5.14.2 Reading and Understanding Code
  - 5.14.3 Fill In the Code
  - 5.14.4 Identifying Errors in Code
  - 5.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

        - 5.14.6 Write a Short Program
        - 5.14.7 Programming Projects
        - 5.14.8 Technical Writing
        - 5.14.9 Group Project (for a group of 1, 2, or 3 students)

Chapter 6 Flow of Control, Part 2: Looping

- Introduction
- 6.1 Event-Controlled Loops Using *while*
- 6.2 General Form for *while* Loops
- 6.3 Event-Controlled Looping
    - 6.3.1 Reading Data from the User
    - 6.3.2 Reading Data from a Text File
- 6.4 Looping Techniques
    - 6.4.1 Accumulation
    - 6.4.2 Counting Items
    - 6.4.3 Calculating an Average
    - 6.4.4 Finding Maximum or Minimum Values
- 6.5 Type-Safe Input Using *Scanner*
- 6.6 Constructing Loop Conditions
- 6.7 Testing Techniques for *while* Loops
- 6.8 Event-Controlled Loops Using *do/while*
- 6.9 Programming Activity 1: Using *while* Loops
    - *Task Instructions*
    - *Troubleshooting*
- 6.10 Count-Controlled Loops Using *for*
    - 6.10.1 Basic Structure of *for* Loops
    - 6.10.2 Constructing *for* Loops
    - 6.10.3 Testing Techniques for *for* Loops
- 6.11 Nested Loops
- 6.12 Programming Activity 2: Using *for* Loops
    - *Instructions*
    - *Troubleshooting*
- 6.13 Chapter Summary
- 6.14 Exercises, Problems, and Projects
    - 6.14.1 Multiple Choice Exercises
    - 6.14.2 Reading and Understanding Code
    - 6.14.3 Fill In the Code

        - 6.14.4 Identifying Errors in Code
        - 6.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
        - 6.14.6 Write a Short Program
        - 6.14.7 Programming Projects
        - 6.14.8 Technical Writing
        - 6.14.9 Group Project (for a group of 1, 2, or 3 students)

Chapter 7 Object-Oriented Programming, Part 2: User-Defined Classes

    - Introduction
    - 7.1 Defining a Class
    - 7.2 Defining Instance Variables
    - 7.3 Writing Class Methods
    - 7.4 Writing Constructors
    - 7.5 Writing Accessor Methods
    - 7.6 Writing Mutator Methods
    - 7.7 Writing Data Manipulation Methods
    - 7.8 Programming Activity 1: Writing a Class Definition, Part 1
    - 7.9 The Object Reference *this*
    - 7.10 The *toString* and *equals* Methods
    - 7.11 *Static* Class Members
    - 7.12 Another Example: The *Rational* Class
    - 7.13 Graphical Objects
    - 7.14 Enumeration Types
    - 7.15 Programming Activity 2: Writing a Class Definition, Part 2
    - 7.16 Creating Packages
    - 7.17 Generating Web-Style Documentation with Javadoc
    - 7.18 Chapter Summary
    - 7.19 Exercises, Problems, and Projects
        - 7.19.1 Multiple Choice Exercises
        - 7.19.2 Reading and Understanding Code
        - 7.19.3 Fill In the Code
        - 7.19.4 Identifying Errors in Code
        - 7.19.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

- 7.19.6 Write a Short Program
- 7.19.7 Programming Projects
- 7.19.8 Technical Writing
- 7.19.9 Group Project (for a group of 1, 2, or 3 students)

Chapter 8 Single-Dimensional Arrays

- Introduction
- 8.1 Declaring and Instantiating Arrays
  - 8.1.1 Declaring Arrays
  - 8.1.2 Instantiating Arrays
  - 8.1.3 Combining the Declaration and Instantiation of Arrays
  - 8.1.4 Assigning Initial Values to Arrays
- 8.2 Accessing Array Elements
- 8.3 Aggregate Array Operations
  - 8.3.1 Printing Array Elements
  - 8.3.2 Reading Data into an Array
  - 8.3.3 Summing the Elements of an Array
  - 8.3.4 Finding Maximum or Minimum Values
  - 8.3.5 Copying Arrays
  - 8.3.6 Changing the Size of an Array
  - 8.3.7 Comparing Arrays for Equality
  - 8.3.8 Displaying Array Data as a Bar Chart
- 8.4 Programming Activity 1: Working with Arrays
    - *Instructions*
    - *Troubleshooting*
- 8.5 Using Arrays in Classes
  - 8.5.1 Using Arrays in User-Defined Classes
  - 8.5.2 Retrieving Command Line Arguments
- 8.6 Searching and Sorting Arrays
  - 8.6.1 Sequential Search of an Unsorted Array
  - 8.6.2 Selection Sort
  - 8.6.3 Insertion Sort
  - 8.6.4 Sorting Arrays of Objects
  - 8.6.5 Sequential Search of a Sorted Array
  - 8.6.6 Binary Search of a Sorted Array
- 8.7 Programming Activity 2: Searching and Sorting Arrays

        - *Instructions*
        - *Troubleshooting*
    - 8.8 Using Arrays as Counters
    - 8.9 Methods Accepting a Variable Number of Arguments
    - 8.10 Chapter Summary
    - 8.11 Exercises, Problems, and Projects
        - 8.11.1 Multiple Choice Exercises
        - 8.11.2 Reading and Understanding Code
        - 8.11.3 Fill In the Code
        - 8.11.4 Identifying Errors in Code
        - 8.11.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
        - 8.11.6 Write a Short Program
        - 8.11.7 Programming Projects
        - 8.11.8 Technical Writing
        - 8.11.9 Group Project (for a group of 1, 2, or 3 students)

Chapter 9 Multidimensional Arrays and the *ArrayList* and *HashMap* Classes
- Introduction
- 9.1 Declaring and Instantiating Multidimensional Arrays
    - 9.1.1 Declaring Multidimensional Arrays
    - 9.1.2 Instantiating Multidimensional Arrays
    - 9.1.3 Combining the Declaration and Instantiation of Multidimensional Arrays
    - 9.1.4 Assigning Initial Values to Multidimensional Arrays
- 9.2 Accessing Multidimensional Array Elements
- 9.3 Aggregate Two-Dimensional Array Operations
    - 9.3.1 Processing All the Elements of a Two-Dimensional Array
    - 9.3.2 Processing a Given Row of a Two-Dimensional Array
    - 9.3.3 Processing a Given Column of a Two-Dimensional Array
    - 9.3.4 Processing a Two-Dimensional Array One Row at a Time
    - 9.3.5 Processing a Two-Dimensional Array One Column at a Time

  - 9.3.6 Displaying Two-Dimensional Array Data as a Bar Chart
- 9.4 Two-Dimensional Arrays Passed to and Returned from Methods
- 9.5 Programming Activity 1: Working with Two-Dimensional Arrays
    - *Instructions*
    - *Troubleshooting*
- 9.6 Other Multidimensional Arrays
- 9.7 The *ArrayList* Class
  - 9.7.1 Declaring and Instantiating *ArrayList* Objects
  - 9.7.2 Methods of the *ArrayList* Class
  - 9.7.3 Looping Through an *ArrayList* Using an Enhanced *for* Loop
  - 9.7.4 Using the *ArrayList* Class in a Program
- 9.8 Programming Activity 2: Working with the *ArrayList* Class
    - *Instructions*
    - *Troubleshooting*
- 9.9 The *HashMap* Class
  - 9.9.1 Declaring and Instantiating *HashMap* Objects
  - 9.9.2 Methods of the *HashMap* Class
  - 9.9.3 Using the *HashMap* in a Program
- 9.10 Chapter Summary
- 9.11 Exercises, Problems, and Projects
  - 9.11.1 Multiple Choice Exercises
  - 9.11.2 Reading and Understanding Code
  - 9.11.3 Fill In the Code
  - 9.11.4 Identifying Errors in Code
  - 9.11.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
  - 9.11.6 Write a Short Program
  - 9.11.7 Programming Projects
  - 9.11.8 Technical Writing
  - 9.11.9 Group Project (for a group of 1, 2, or 3 students)

Chapter 10 Object-Oriented Programming, Part 3: Inheritance, Polymorphism, and Interfaces

Introduction
10.1 Inheritance
10.2 Inheritance Design
  10.2.1 Inherited Members of a Class
  10.2.2 Subclass Constructors
  10.2.3 Adding Specialization to the Subclass
  10.2.4 Overriding Inherited Methods
10.3 The *protected* Access Modifier
10.4 Programming Activity 1: Using Inheritance
    Instructions
10.5 *Abstract* Classes and Methods
10.6 Polymorphism
10.7 Programming Activity 2: Using Polymorphism
    Instructions
10.8 Interfaces
10.9 Chapter Summary
10.10 Exercises, Problems, and Projects
  10.10.1 Multiple Choice Exercises
  10.10.2 Reading and Understanding Code
  10.10.3 Fill In the Code
  10.10.4 Identifying Errors in Code
  10.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
  10.10.6 Write a Short Program
  10.10.7 Programming Projects
  10.10.8 Technical Writing
  10.10.9 Group Project (for a group of 1, 2, or 3 students)

Chapter 11 Exceptions and Input/Output Operations
Introduction
11.1 Simple Exception Handling
11.2 Catching Multiple Exceptions
11.3 Reading Text Files Using *Scanner*
11.4 The *java.io* Package
11.5 Recovering from an Exception
11.6 Writing and Appending to Text Files

  - 11.6.1 Writing to Text Files
  - 11.6.2 Appending to Text Files
- 11.7 Reading Structured Text Files
  - 11.7.1 Parsing a *String* Using *Scanner*
  - 11.7.2 Reading Structured Data Using *Scanner*
- 11.8 Programming Activity 1: Reading from a Structured Text File
    - *Instructions*
    - *If you have time*
    - *Troubleshooting*
- 11.9 Streams
- 11.10 Reading Formatted Open Data from a Remote Location
  - 11.10.1 Accessing Remote Data
  - 11.10.2 JSON Formatting and Parsing
  - 11.10.3 Reading, Parsing, Streaming, and Processing Remote Data
- 11.11 Reading and Writing Objects to a File
  - 11.11.1 Writing Objects to Files
  - 11.11.2 Reading Objects from Files
- 11.12 Programming Activity 2: Reading Objects from a File
    - *Task Instructions: Reading from the transactions.obj File*
    - *If you have time*
    - *Troubleshooting*
- 11.13 User-Defined Exceptions
- 11.14 Chapter Summary
- 11.15 Exercises, Problems, and Projects
  - 11.15.1 Multiple Choice Exercises
  - 11.15.2 Reading and Understanding Code
  - 11.15.3 Fill In the Code
  - 11.15.4 Identifying Errors in Code
  - 11.15.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
  - 11.15.6 Write a Short Program
  - 11.15.7 Programming Projects
  - 11.15.8 Technical Writing
  - 11.15.9 Group Project (for groups of 2, 3, or more

students)

Chapter 12 Recursion
- Introduction
- 12.1 Simple Recursion: Identifying the General and Base Cases
- 12.2 Recursion with a Return Value
  - 12.2.1 Computing the Factorial of a Number
  - 12.2.2 Computing the Greatest Common Divisor
- 12.3 Recursion with Two Base Cases
- 12.4 Programming Activity 1: Checking for a Palindrome
    - *Instructions*
    - *Task Instructions*
    - *Troubleshooting*
- 12.5 Simple Recursion on an Array
- 12.6 Binary Search: A Recursive Solution
- 12.7 Programming Activity 2: The Towers of Hanoi
    - *Instructions*
    - *Task Instructions*
    - *Troubleshooting*
- 12.8 Merge Sort
- 12.9 Quick Sort
- 12.10 Recursion Versus Iteration
- 12.11 Chapter Summary
- 12.12 Exercises, Problems, and Projects
  - 12.12.1 Multiple Choice Exercises
  - 12.12.2 Reading and Understanding Code
  - 12.12.3 Fill In the Code
  - 12.12.4 Identifying Errors in Code
  - 12.12.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
  - 12.12.6 Write a Short Program
  - 12.12.7 Programming Projects
  - 12.12.8 Technical Writing
  - 12.12.9 Group Projects (for a group of 1, 2, or 3 students)

Chapter 13 An Introduction to Data Structures
- Introduction
- 13.1 Linked Lists
  - 13.1.1 Linked-List Concepts and Structure
  - 13.1.2 Linked-List Basics
  - 13.1.3 Methods of a Linked List
  - 13.1.4 Testing a Linked-List Class
- 13.2 Linked Lists of Objects
  - 13.2.1 A Linked-List Shell
  - 13.2.2 Generating an Exception
  - 13.2.3 Other Methods of a Linked List
  - 13.2.4 Testing a Linked-List Class
- 13.3 Implementing a Stack Using a Linked List
- 13.4 Implementing a Queue Using a Linked List
- 13.5 Array Representation of Stacks
- 13.6 Programming Activity 1: Writing Methods for a Stack Class
    - *Instructions*
    - *Troubleshooting*
- 13.7 Array Representation of Queues
- 13.8 Sorted Linked Lists
- 13.9 Programming Activity 2: Writing *Insert* and *Delete* Methods for a Sorted Linked List
    - *Instructions*
    - *Troubleshooting*
- 13.10 Doubly Linked Lists
- 13.11 Linked Lists Using Generic Types
- 13.12 Recursively Defined Linked Lists
- 13.13 Chapter Summary
- 13.14 Exercises, Problems, and Projects
  - 13.14.1 Multiple Choice Exercises
  - 13.14.2 Reading and Understanding Code
  - 13.14.3 Fill In the Code
  - 13.14.4 Identifying Errors in Code
  - 13.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
  - 13.14.6 Write a Short Program

13.14.7 Programming Projects
13.14.8 Technical Writing
13.14.9 Group Project (for a group of 1, 2, or 3 students)

Chapter 14 Running Time Analysis
Introduction
14.1 Orders of Magnitude and Big-Oh Notation
14.2 Running Time Analysis of Algorithms: Counting Statements
14.3 Running Time Analysis of Algorithms and Impact of Coding: Evaluating Recursive Methods
Handwaving Method
Iterative Method
Proof by Induction Method
Other Methods
14.4 Programming Activity: Tracking How Many Statements Are Executed by a Method
Instructions
Troubleshooting
14.5 Running Time Analysis of Searching and Sorting Algorithms
14.6 Simulation of Various Sorting Algorithms
14.7 Chapter Summary
14.8 Exercises, Problems, and Projects
14.8.1 Multiple Choice Exercises
14.8.2 Compute the Running Time of a Method
14.8.3 Programming Projects
14.8.4 Technical Writing
14.8.5 Group Project (for a group of 1, 2, or 3 students)

Chapter 15 General Trees, Binary Trees, and Binary Search Trees
Introduction
15.1 Trees
15.2 Binary Trees
15.3 Binary Tree Implementation
15.4 Programming Activity 1: Working with Binary Trees

Instructions

Troubleshooting

15.5 Binary Search Trees

15.5.1 Searching in a Binary Search Tree

15.5.2 Inserting in a Binary Search Tree

15.5.3 Deleting from a Binary Search Tree

15.6 Binary Search Tree Implementation

15.7 Rebalancing a Binary Search Tree

15.8 Programming Activity 2: Working with Binary Search Trees

Instructions

Troubleshooting

15.9 Chapter Summary

15.10 Exercises, Problems, and Projects

15.10.1 Multiple Choice Exercises

15.10.2 Reading and Understanding Code

15.10.3 Fill In the Code

15.10.4 Identifying Errors in Code

15.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

15.10.6 Write a Short Program

15.10.7 Programming Projects

15.10.8 Technical Writing

15.10.9 Group Project (for a group of 1,2, or 3 students)

Chapter 16 Graphical User Interfaces (available online in the eBook)

Chapter 17 Graphical User Interfaces Using JavaFX (available online in the eBook)

Appendix A Java Reserved Words and Keywords

Appendix B Operator Precedence

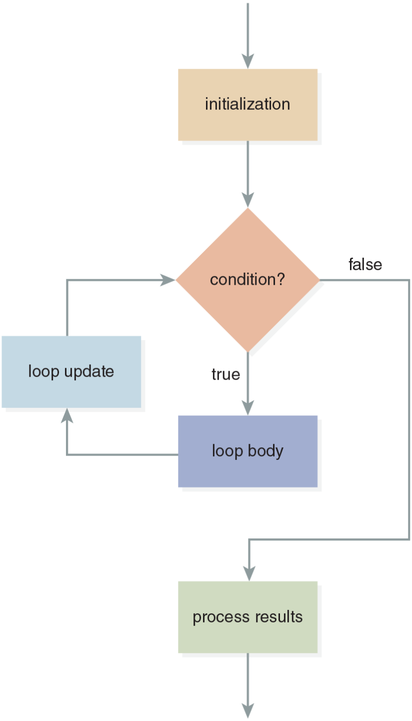
Appendix C The Unicode Character Set

Appendix D Representing Negative Integers

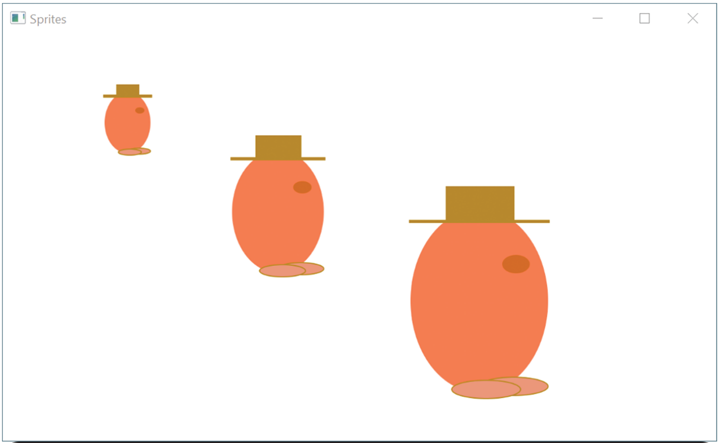
Appendix E Representing Floating-Point Numbers (available online in the eBook)

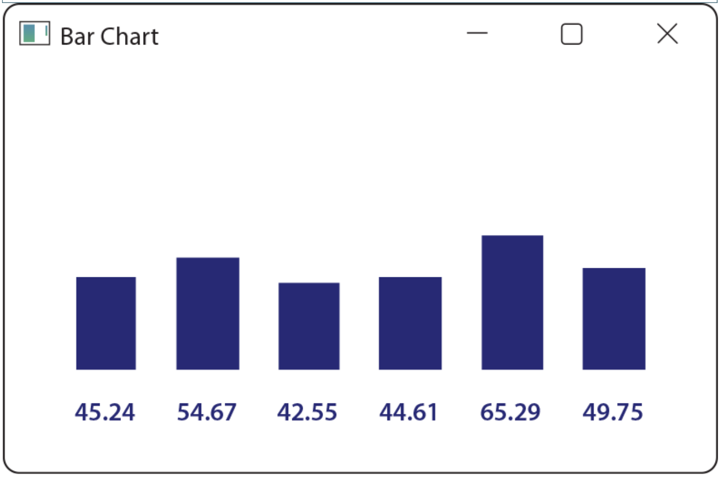
Appendix F Solutions to Selected Exercises (available online in the

eBook)
Index


initialization
condition?
false
true
loop update
loop body
process results

Look
It
Up


Sprites


Bar Chart
45.24
54.67
42.55
44.61
65.29
49.75

Python

is used for single line comments

```python
# This is a comment
```

Java

// is used for single line comments

```java
// This is a comment
```

Python

''' and ''' are used to delimit multiple line comments

```python
''' This is a
    comment that spans over several
   lines
'''
```

Java

/* and */ are used to delimit multiple line comments

```java
/* This is a
    comment that spans over several lines
*/
```

Python

There is no need to end a statement with a semicolon (;).

Data types are implicit. We do not need to declare a variable, we just use it.

```python
age = 18
gpa = 3.7
```

Java

Every statement ends with a semicolon (;).

Data types are explicit. A variable must be declared before using it.

```java
int age;
age = 18;
double gpa = 3.7;
```

Python

/ is used floating point division, // is used for integer division

```python
average = 373 / 4
# floating point division
quarters = 102 // 25
# integer division
```

Java

The type of division (floating point or integer) is based on the data types of the two operands.

```java
double average = 373.0 / 4;
// floating point division
int quarters = 102 / 25;
// integer division
```

Python

Single quotes and double quotes have the same meaning.

```python
grade = 'A'
city = "New York"
```

Java

Single quotes are used for characters, double quotes are used for Strings.

```java
char grade = 'A';
String city = "New York";
```

Python

There is no formal concept of a constant, but using uppercase for a variable is good practice to denote that we should not change its value.

```python
DAYS = 365
# Please do not change the
# value of DAYS
```

Java

The *final* keyword is used to declare constants. The value of a constant cannot be changed.

```java
final int DAYS = 365;
// DAYS' value cannot be
// changed
```

Python

Indentation counts. Parentheses around the condition are not needed, and a colon (:) precedes the indented statements.

```python
if age >= 18:
    print( "You can vote" )
    print( "Congratulations" )
```

Java

Indentation does not count, but we use it for style and readability. Parentheses are required around conditions. If there is more than one statement attached to a selection or looping construct, we use curly braces for the block.

```java
if ( age >= 18 )
{
    System.out.println( "You can vote" );
    System.out.println( "Congratulations" );
}
```

Python

The logical operators are *and*, *or*, and *not*.

```python
if age >= 18 and age <= 65:
    print( "Hi" )
```

Java

The logical operators are *&&* (for and), *//* (for or), and *!* (for not).

```java
if ( age >= 18 and age <= 65 )
    System.out.println( "Hi" );
```

Python

The keyword *def* is used for functions. The parameter list is a comma delimited list of variable names. The function header is followed by a colon (:). The body of the function is indented.

```python
def sum( a, b ):
   return a + b
```

Java

The parameter list of a function (called method in Java) is a comma delimited list of variable names and their respective data types. The method header includes the return data type (void if the method does not return anything). The body of the method is inside a set of curly braces. Indentation is optional but recommended for better readability.

```java
int sum( int a, int b )
{
  return a + b;
}
```

© Lois GoBe/Shutterstock.

Preface

Purpose of This Text and Its Audience

Java Illuminated, Sixth Edition, covers all of the material required for the successful completion of an introductory course in Java. While the focus is on the material required for the Computer Science I (CS1) and Computer Science II (CS2) curricula, students enrolled in Information Systems, Information Technology, or self-directed study courses will find the text useful as well. It has been written to provide introductory computer science students with a comprehensive overview of the fundamentals of programming using Java as the teaching language. In addition, the text presents other topics of interest, including graphical user interfaces (GUI), data structures, file input and output, and graphical applications.

Throughout the text, we take an “active learning” approach to presenting the material. Instead of merely presenting the concepts to students in a one-sided, rote manner, we ask them to take an active role in their understanding of the concepts through the use of numerous interactive examples, exercises, and projects.

Coverage and Approach

Our approach is to teach object-oriented programming in a progressive manner. We start in Chapter 1 by presenting an overview of object-oriented programming. In Chapter 3, we delve a little deeper into the concepts of classes and objects and introduce the student to many of the useful classes in the Java Class Library. Our emphasis at this point is on using classes; we teach the student how to read APIs in order to determine how to instantiate objects and call methods of the classes. In Chapter 7, we move on to designing user-defined classes, and in Chapter 10, we present inheritance, polymorphism, and interfaces. Throughout the text, we present concepts in an object-oriented context.

Our philosophy is to emphasize good software engineering practices by focusing on designing and writing correct, maintainable programs. As such, we discuss pseudocode, testing techniques, design trade-offs, and other software engineering tips.

We teach the student basic programming techniques, such as accumulation, counting, calculating an average, finding maximum and minimum values, using flag and toggle variables, and basic searching and sorting algorithms. In doing so, we emphasize the patterns inherent in programming. Concepts are taught first, followed by fully implemented examples with source code. We promote Java standards, conventions, and methodologies.

What's New in *Java Illuminated*

In this edition, we have incorporated the latest Java features as they apply to this text: text blocks, the new *switch* statement syntax and *switch* expression, the new syntax for *instanceof*, and the *record* class. We have added coverage of merge sort, quick sort, and the *HashMap* class. We have also eliminated the *SimpleDate* class and incorporated more materials into our coverage of calling methods and building a class in Chapters 3 (using classes) and 7 (creating our own classes).

For students coming from Python, we have added a small section at the beginning of the book showing examples of Python syntax and its Java equivalent.

Throughout the text, we have updated, improved, or replaced examples accordingly.

In Chapter 1, we added how to run a program in one step, without compiling, and how to retrieve the version of *javac* and *java*.

In Chapter 2, we show how to use text blocks, for example, to display a menu.

In Chapter 3, we have eliminated the *SimpleDate* class. We use the *String* class to explain how to instantiate objects and call methods. We have added more examples using the *String* class, such as parsing an email address.

In Chapter 5, we explain the new *switch* syntax and how to use *switch* expressions.

In Chapter 7, in addition to the *Auto* class, we have added a new example: a *Rational* class. We added explanations showing how *this* is the object reference that called a method. We also cover the new syntax of the *instanceof* operator.

In Chapter 9, we include coverage of the *HashMap* class.

In Chapter 11, we have converted some of the examples to using the *record* class and explain how it works well if all we need is a class to store data (that we read from a file). We also changed the example reading data from a remote location to a simpler example.

In Chapter 12 (recursion, formerly Chapter 13), we have added merge sort and quick sort, including examples and explanations.

In Chapter 13 (linked lists, formerly Chapter 14), we are now using the *compareTo* method in the sorted linked list example.

In Chapter 14 (running time analysis, formerly Chapter 15), we have added simulation code counting comparison statements for insertion sort, merge sort, and quick sort, including when the array is already sorted.

Chapter 15 (binary trees and binary search trees) is a new chapter in this text, replacing the old Chapter 12 (graphical user interfaces), which has been moved online. We start with general trees and tree concepts (e.g., root, height, level, and leaves), then cover binary trees, including pre order, in order, and post order traversals. We discuss the motivation for binary search trees and cover binary search trees, including an analysis of running times for search, insert, and delete methods. We also cover the need to rebalance the tree and implement and discuss a rebalancing recursive method.

Learning Features

Recognizing today's students' growing interest in visualization, we distribute techniques for producing graphical output throughout the book, starting in Chapter 4 with graphical applications. An example using either animation or graphical output is included in most chapters. Instructors who are not interested in incorporating graphics into their curriculum can simply skip these sections. In addition, some of our examples are small games, which we find motivational for students.

In each chapter, we include one or two Programming Activities, which are designed to provide visual feedback to the students so that they can assess the correctness of their code. In most Programming Activities, we provide a framework, usually with a graphical user interface, to which the student adds code to complete the application. The student should be able to finish the Programming Activity in about 15 to 20 minutes; thus, these activities can be used in the classroom to reinforce the topics just presented. Each Programming Activity also includes several discussion questions that test the student's understanding of the concepts the activity illustrates. The Programming Activities are also appropriate for a closed or open laboratory environment. In short, this text can be used in a traditional lecture environment, a computer-equipped classroom, or a lab environment.

We also provide the instructor and students with an extensive variety of end-of-chapter material: multiple-choice questions, examples that ask the student to predict the output of prewritten code or to fill in missing code, debugging activities, short exercises, programming projects, technical writing assignments, one or more learning-to-learn exercises called Look It Up, and a higher-difficulty group project.

Chapter-by-Chapter Overview

The chapters are logically organized from simple to more difficult topics, while incorporating object orientation as needed, taking into account the specifics of the Java language. Here is a brief summary of the topics covered in each chapter.

Chapter 1: Introduction to Programming and the Java Language

We introduce the student to the concept of programming, first covering computer hardware and operating systems, and following with a brief evolution of programming languages, including an introduction to object-oriented programming. We explain programming basics and pseudocode as a program design technique. The student writes, compiles, and debugs their first program using an integrated development environment.

Chapter 2: Programming Building Blocks—Java Basics

In this chapter, we concentrate on working with variables and constants of primitive data types and composing arithmetic expressions. We illustrate the differences between integer and floating-point calculations and introduce operator precedence. We introduce jshell, the Java programming sandbox. We use a combination of jshell sessions and complete programs to demonstrate the concepts at hand.

Chapter 3: Object-Oriented Programming, Part 1: Using Classes

Chapter 3 introduces classes from the user, or client, standpoint and discusses the benefits of encapsulation and code reuse. The student learns how to instantiate objects and call methods. We also demonstrate useful Java classes for console input and output, formatting output, performing mathematical calculations, generating random numbers, and using methods of the Wrapper classes. We use methods of the *String* class to visually illustrate the passing of arguments and receiving of and use of return values.

Chapter 4: Introduction to Graphical Applications

Chapter 4 presents several methods of the JavaFX *GraphicsContext* class that can be used to create graphical output by drawing shapes, text, and sprites. The windowing graphics coordinate system is explained and using color is also explored. We demonstrate these graphics methods in JavaFX applications. Instructors wishing to postpone or skip graphics coverage altogether can use as little or as much of this chapter as they desire.

Chapter 5: Flow of Control, Part 1: Selection

Various forms of the *if, if/else*, and *if/else if* statements are presented, along with the appropriate situations in which to use each form. We also demonstrate nested *if/else* statements and testing techniques. We begin our coverage of scope by introducing block scope. Later chapters build upon this foundation. As part of our object-oriented programming coverage, we teach the importance of comparing objects using the *equals* method. This chapter also covers the conditional operator, the *switch* statement, and *switch* expressions.

Chapter 6: Flow of Control, Part 2: Looping

This is probably the most important chapter in the text. We have found that looping and repetition are the most difficult basic programming concepts for the average student to grasp. We try to ease the student's understanding of looping techniques by presenting patterns to follow in coding basic algorithms: accumulation, counting, calculating an average, and finding minimum and maximum values. Looping is further explored as a tool for validation of input values. We also introduce toggle variables and flag variables as tools to facilitate writing loops. We continue our coverage of scope by illustrating the scope of variables declared within the *while* loop body and *for* loop header. We concentrate on using the *while* loop *for* event-controlled and sentinel-controlled repetition and the *for* loop for count-controlled looping. A large section focuses on constructing loop conditions, which is often a challenging task for the student. Sections are also provided on testing techniques for loops. In this chapter, we also introduce reading data from a text file using the *Scanner* class.

Chapter 7: Object-Oriented Programming, Part 2: User-Defined Classes

In this chapter, we teach the student to write classes, as well as client applications that instantiate objects and call methods of the class. We present class design techniques and standard patterns for writing constructors, mutators and accessors, and the *toString, equals*, and other user-defined methods. We further explain scope in the context of class members and method parameters. We explain and show that *this* refers to the object reference calling an instance method of a class. We also explain how to write classes with *static* methods and *static* data. Additionally, we illustrate how to write methods so that the method calls can be chained. *Enum* is also covered as a user-defined class type. Finally, we teach the student how to use Javadoc.

Chapter 8: Single-Dimensional Arrays

This chapter begins with the declaration, instantiation, and initialization of single-dimensional arrays. From there, the student learns to perform the basic programming techniques (accumulation, counting, calculating an average, and finding maximum and minimum values) on array elements. We also cover arrays as instance variables of a class, and demonstrate maintaining encapsulation while accepting arrays as method parameters and returning arrays from methods. We demonstrate how to represent an array as a bar chart. Basic searching and sorting algorithms are also presented, including sequential and binary searches and Selection and Insertion sorts.

Chapter 9: Multidimensional Arrays and the *ArrayList* and *HashMap* Classes

We focus in this chapter on two-dimensional array processing, including techniques for processing the elements in the entire array, or the elements in a specific column or row. We also demonstrate the extra processing needed to handle arrays with rows of different lengths. Creating a bar chart of the data in the array is also demonstrated. In addition, we extrapolate the concepts from two-dimensional arrays to discuss multidimensional arrays.

We present the *ArrayList* class as an expandable array and demonstrate using classes with generic types, the enhanced *for* loop, and autoboxing and unboxing. We also cover the *HashMap* class.

Chapter 10: Object-Oriented Programming, Part 3: Inheritance, Polymorphism, and Interfaces

Continuing our object-oriented programming coverage, we discuss the important concepts and benefits of inheritance and the design of class hierarchies, including abstract classes. We cover inherited members of a class, constructing objects of a subclass, adding specialization to a subclass, overriding inherited methods, and calling methods of the superclass. We discuss the trade-offs of declaring members as *protected* versus *private*. We demonstrate polymorphism with a graphical example. We introduce the student to interfaces, emphasizing code reuse as the motivation for defining interfaces. We explain *default* methods and how we can use them in new versions of a program so that our old code does not break as we add new features. We also demonstrate polymorphism using interfaces. Interfaces are used extensively in Graphical User Interfaces (see online chapter). We introduce UML diagrams to help students visualize the inheritance hierarchy of the examples.

Chapter 11: Exceptions and Input/Output Operations

Recognizing that building robust applications requires error handling, we present exception handling as a tool for validating user input and recovering from errors at run time. We demonstrate handling predefined exceptions and writing user-defined exceptions.

With this knowledge, the student is ready to perform file input and output operations. We demonstrate reading and writing *Strings* and primitive data types to text files, and reading and writing objects directly to files. The *Scanner* class is used to read and parse input from text files and *Strings*. We use *record* classes to read data into objects of these *record* classes. We demonstrate how to read and place file contents into a *Stream* and how to filter and process that *Stream*. We use lambda expressions when processing the *Stream*. We use the *org.json* package to show how to parse JSON data. We demonstrate how to read data from a remote file located on the Internet, including parsing JSON data and converting that data into a *Stream* for filtering and processing.

Chapter 12: Recursion

Recursion is presented as a design technique, reducing the size of a problem until an easy-to-solve problem is reached. We demonstrate recursive methods with one base case and with multiple base cases, and with and without return values. Specific examples provided include computing the factorial of a number, finding the greatest common divisor, performing a binary search, determining if a phrase is a palindrome, calculating combinations, solving the Towers of Hanoi problem, merge sort, and quick sort. We use the *StackWalker* class to demonstrate how values are pushed onto and popped off the stack as recursive calls are executed. The benefits and trade-offs of recursion versus iteration are also discussed.

Chapter 13: An Introduction to Data Structures

In this chapter, we cover data structures by exploring the concepts and implementations of various types of linked lists, stacks, and queues. We demonstrate many types and uses of linked lists: a singly linked list, a linked list as a stack, a linked list as a queue, a doubly linked list, a sorted linked list, and a recursively defined linked list. Arrays as stacks and circular arrays as queues are also covered in detail.

We begin with a list of primitive types (*int*) and progress to a list consisting of objects of a user-defined *Player* class. Then we cover defining a class using generic types to demonstrate how a list can be defined to hold generic objects.

Chapter 14: Running Time Analysis

We explain how to evaluate the performance of an algorithm in this chapter. We explain the Big-Oh notation and orders of magnitude. Students learn various methods for deriving performance estimates: counting statements in loops, iterative, handwaving, and proof by induction analyses for recursive methods. We demonstrate how the coding of an algorithm influences its running time. Worst-case, best-case, and average-case performances are explained and illustrated. We show an example counting comparison statements for insertion sort, merge sort, and quick sort, included when the array is already sorted, and how that relates to the math analysis of the running times of these methods.

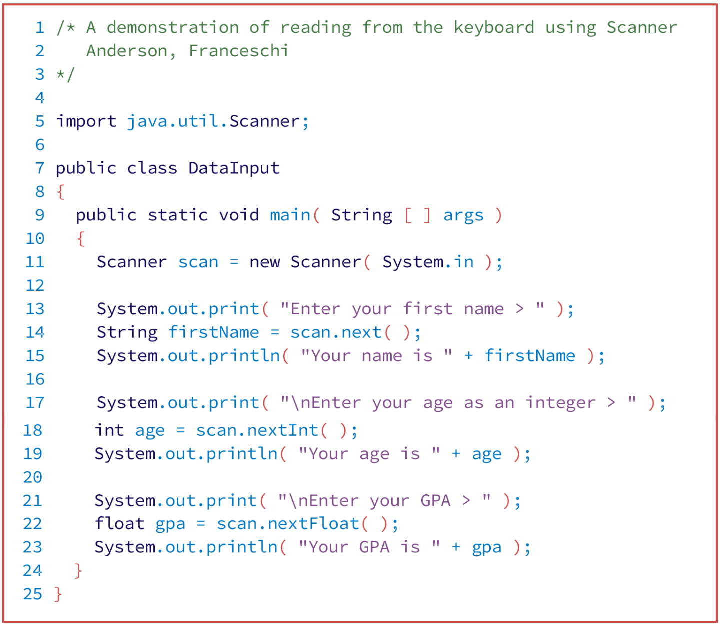
Chapter 15: General Trees, Binary Trees, and Binary Search Trees

We cover general trees and tree concepts (e.g., root, height, level, leaves, complete and full binary trees). Then, we cover binary trees, including pre order, in order, and post order traversals. We discuss the desirability of trees to be compact and compute the number of levels in a full binary tree as a function of the number of nodes in the tree. We discuss the motivation for binary search trees and cover binary search trees, including search, insert, and delete methods. Then, we analyze the running times of these methods. We also discuss the need to rebalance the tree and implement and discuss a rebalancing recursive method.

Pedagogy

Concepts are always taught first, followed by complete, executable examples illustrating these concepts. Most examples demonstrate real-life applications so that the student can understand the need for the concept at hand. The example code is colored to better illustrate the syntax of the code and to reflect the use of colors in today's IDE tools, as shown in this example from Chapter 3:

Example 3.9 Reading from the Keyboard Using *Scanner*



```java
/* A demonstration of reading from the keyboard using Scanner
   Anderson, Franceschi
*/

import java.util.Scanner;

public class DataInput
{
  public static void main( String [ ] args )
  {
    Scanner scan = new Scanner( System.in );

    System.out.print( "Enter your first name > " );
    String firstName = scan.next( );
    System.out.println( "Your name is " + firstName );

    System.out.print( "\nEnter your age as an integer > " );
    int age = scan.nextInt( );
    System.out.println( "Your age is " + age );

    System.out.print( "\nEnter your GPA > " );
    float gpa = scan.nextFloat( );
    System.out.println( "Your GPA is " + gpa );
  }
}
```

Description

Note that in this example and throughout the text, we place the opening curly brace of a block on a line of its own. We have found this placement easier for a student new to Java to understand where a block begins and ends.

Figures and tables are used to illustrate or summarize the concept at hand, such as these from Chapters 6 and 7:

Figure 6.1 Flow of Control of a *while* Loop

Description

Access Modifiers

Access Modifier	**Class or Member Can Be Referenced By ...**
public	methods of the same class, as well as methods of other classes
private	methods of the same class only
protected	methods in the same class, as well

	as methods of subclasses and methods in classes in the same package
no modifier (package access)	methods in the same package or same folder only

Description

In each chapter, we emphasize good design concepts using “Software Engineering Tips,” such as the one to the right from Chapter 7.

SOFTWARE ENGINEERING TIP

Define instance variables of a class as *private* so that only methods of the class will be able to set or change their values.

COMMON ERROR TRAP

Be sure that both operands of the logical AND and logical OR operators are *boolean* expressions. Expressions such as `x < y && z`, with *x, y,* and *z* being numeric types, are illegal. Instead, use the expression

x < y && x < z

We also provide “Common Error Traps,” such as the one to the right from Chapter 5, to alert students against common syntax and logic errors.

In each chapter, “active learning” Programming Activities reinforce concepts with enjoyable, hands-on projects that provide visual feedback to the students. These activities can be done in lab-type classrooms or can be assigned as projects. A header for a Programming Activity looks like this:

6.9 Programming Activity 1: Using *while* Loops

In this activity, you will work with a sentinel-controlled *while* loop, performing this activity:

> Write a *while* loop to process the contents of a grocery cart and calculate the total price of the items. It is important to understand that, in this example, we do not know how many items are in the cart.

Because technology is ever-changing, students need to be able to learn new concepts and find relevant classes on their own. To promote the development of this important skill of “learning-to-learn,” we include one or more “Look It Up” questions in the end-of-chapter exercises. These questions require the student to perform some research to acquire the knowledge necessary to answer the question and write the required code.

Description

Within the online resources, for each chapter, we provide a movie that illustrates the execution of code implementing the concepts taught in the chapter. Each movie animates a brief code sample, one line at a time, and is controlled by the user via a “Next Step” button. These modules can be beneficial for students who learn best with visual aids, graphs, illustrations, and at their own pace outside the classroom. The modules are announced in each chapter using a special icon as in the sample above.

CODE IN ACTION

To see two step-by-step illustrations of *do/while* loops, look for this chapter’s movie within the online resources. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

Graphics Coverage

Graphics are distributed throughout the book and are used to engage the student and reinforce the chapter concepts. The graphics coordinate system, methods for drawing shapes and text, and color concepts are presented with simple JavaFX applications in Chapter 4. At least one graphical example is presented in subsequent chapters using JavaFX. In Chapter 5, a *switch* statement is used to draw a smiley face; in Chapter 6, a loop is used to draw a bull’s eye target. Classes for displayable objects are presented in Chapter 7; drawing a bar chart of array data is illustrated in Chapters 8 and 9; polymorphism is demonstrated using a Tortoise and Hare Race in Chapter 10; GUIs, dynamic charts, and JavaFX animations are covered in online chapters. The two figures that follow illustrate graphical examples from Chapters 7 and 8.

Figure 7.12 The *SpriteClient* Window

Description

Figure 8.15 The *cellBills* Array as a Bar Chart

End-of-Chapter Exercises and Problems

A large collection of exercises and problems is proposed at the end of each chapter. Short exercises cover programming from a variety of angles: multiple choice concept questions, reading and understanding code segments, filling in some code, correcting errors, and interpreting compiler error messages to diagnose application bugs. Many programming projects are proposed with an emphasis on writing *classes*, not just a *program*. A more challenging group project is proposed in each chapter, allowing students to work as a group and develop communication skills, in accordance with recommendations from accreditation organizations and potential employers in the computer industry. Small, essay-type questions are also proposed to enable students to acquire proficiency in technical writing and communication.

Web Materials Accompanying This Book

Available on the Jones & Bartlett Learning website (www.jblearning.com):

- Programming Activity framework code
- Full example code from each chapter
- Swing versions of Chapters 4 and GUI chapters using JavaFX and Swing.

Appendices

The appendices include the following:

- Java reserved words and keywords
- Operator precedence
- Unicode character set
- Representing negative numbers
- Representing floating-point numbers
- Answers to selected exercises

Instructor Resources

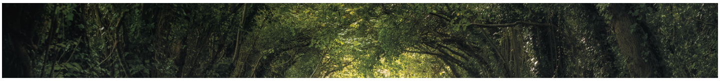
These materials are available to instructors on the Jones & Bartlett Learning website (go.jblearning.com/javailluminated6e), and include

- Programming activity solution code (for instructors only)
- Answers to many end-of-chapter exercises
- Lecture slides in PowerPoint format for each chapter
- Test banks for each chapter

Contacting the Authors

We have checked and rechecked the many technical details in this text. Despite our best efforts, however, we realize that some errors may have been missed. If you discover a technical error in the text, please contact us at hervefranceschi@gmail.com. We will post any corrections on the text's website: go.jblearning.com/javailluminated6e.

Python vs. Java: A Few Simple Concepts



© Lois GoBe/Shutterstock.

Acknowledgments

We would like to acknowledge the contributions of many partners, colleagues, and family members to this book.

First and foremost, we would like to thank our publisher, Jones & Bartlett Learning, especially Edward Hinman, Product Manager; Melissa Duffy, Content Strategist; and Jennifer Risden, Project Manager. We also want to thank the compositor, Straive; the proofreader, Charlotte Zuccarini; and Briana Yates, who designed the cover.

Second, we extend our thanks to the reviewers: Zareh Gorjian, Pasadena City College; Lee Nicholson, Alpharetta High School; Jody Paul, Metropolitan State University of Denver; Steven Kreutzer, Bloomfield College; Larry Henderson, Wenatchee Valley College; Fred D'Angelo, Pima Community College; Waleed Farag, Indiana University of Pennsylvania; Jason Smith, University of Texas, Dallas; Valerie Chau, Palomar College; Jeanann Boyce, Montgomery College; Kevin Brunner, Graceland University; Patrick Plunkett, Wheeling Jesuit University; Russel Bruhn, University of Arkansas, Little Rock; Burdett E. Wilson, Macon Area Career and Technical Education Center; Jonathan Sprague, College of Lake County; Giuseppe Turini, Kettering University; Nary Subramanian, University of Texas, Tyler; Jeffrey Kimball, Southwest Baptist University; Zhiling Lan, Illinois Institute of Technology; Peter L. Stanchev, Kettering University; Jacquelyne Lewis, North Carolina Wesleyan College; Vicky Hardin, Jefferson Community & Technical College; Jesse Kidd, Motlow State Community College; Mark Meysenburg, Doane College; Derrf Seitz, Georgia Military College; Ellen Spertus, Mills College; Scott Reed, College of Lake County; Peter Johnson, South Central College; and Sungbum Hong, Jackson State University. We have taken your thoughtful comments to heart and we think the book is better for them.

Julie Anderson would also like to acknowledge the pedagogical insight of Richard Rasala and Viera Proulx of Northeastern University. Thanks also to Jon Dornback, Garth Gerstein, and our former colleagues Pat Smit and the late Earl Gottsman.

I am extremely grateful for the help extended by many family members: my father, Glenn Austin, sons Brian and Jon Anderson, daughter-in-law Silvia Eckert, grandson, Ben Anderson (for his sound effects), sister Kathleen Austin, and mother-in-law Virginia Anderson. And of course, much gratitude goes to my loving husband, Tom, for his support and encouragement.

—Julie Anderson

I also recognize the support of my family. In particular, my brother, Paul, provided feedback on our sample chapter and the movies, and my wife, Kristin, gave her support and provided advice.

—Hervé Franceschi

© Lois GoBe/Shutterstock.

CHAPTER 1

Introduction to Programming and the Java Language

CHAPTER CONTENTS

Introduction
1.1 Basic Computer Concepts
    1.1.1 Hardware
    1.1.2 Operating Systems
    1.1.3 Application Software
    1.1.4 Computer Networks and the Internet
1.2 **Practice Activity: Displaying System Configuration**
    1.2.1 Displaying Windows Configuration Information
    1.2.2 Displaying Mac OS Configuration Information
1.3 Data Representation
    1.3.1 Binary Numbers
    1.3.2 Using Hexadecimal Numbers to Represent Binary

    Numbers
    1.3.3 Representing Characters with the Unicode Character Set

1.4 Programming Languages

    1.4.1 High- and Low-Level Languages
    1.4.2 An Introduction to Object- Oriented Programming
    1.4.3 The Java Language

1.5 An Introduction to Programming

    1.5.1 Programming Basics
    1.5.2 Program Design with Pseudocode
    1.5.3 Developing a Java Application
    1.5.4 **Programming Activity 1: Writing a First Java Application**
    1.5.5 Making a JAR File

1.6 Chapter Summary
1.7 Exercises, Problems, and Projects

    1.7.1 Multiple Choice Exercises
    1.7.2 Converting Numbers
    1.7.3 General Questions
    1.7.4 Technical Writing
    1.7.5 Group Project

Introduction

Computer applications touch almost every aspect of our lives. They run automated teller machines, the grocery store's checkout register, the appointment calendar at your doctor's office, airport kiosks for flight check-in, a restaurant's meal-ordering system, and online auctions, just to name a few applications. On your personal computer, you may run a word processor, virus detection software, a spreadsheet, computer games, and an image processing system.

Someone, usually a team of programmers, wrote those applications. If you're reading this text, you're probably curious about what's involved in writing applications, and you would like to write a few yourself. Perhaps you have an idea for the world's next great application or computer game.

In this text, we'll cover the basics of writing applications. Specifically, we'll use the Java programming language. Keep in mind, however, that becoming a good programmer requires more than mastering the rules, or **syntax**, of a programming language. You also must master basic programming techniques. These are established methods for performing common programming operations, such as calculating a total, finding an average, or arranging a group of items in order.

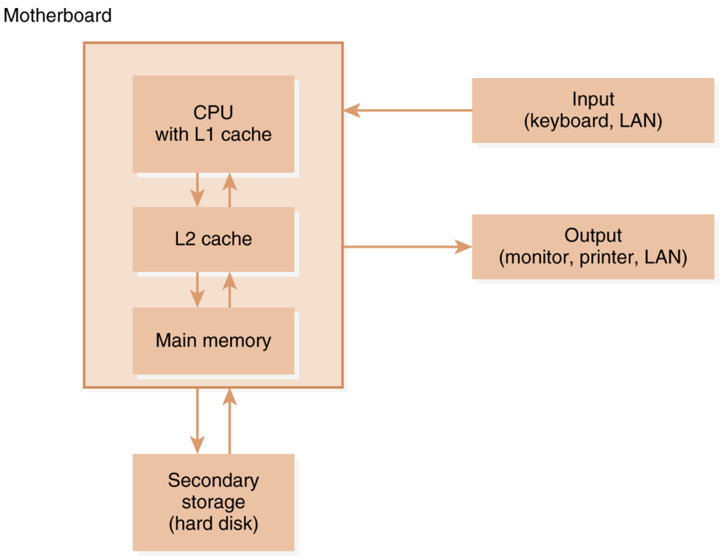
You also must master good software engineering principles so that you design code that is readable, easily maintained, and reusable. By readable, we mean that someone else should be able to read your program and figure out what it does and how it does it. Writing readable code is especially important for programmers who want to advance in their careers, because it allows someone else to take over the maintenance of your program while you move on to bigger and better responsibilities. Ease of maintenance is also an important aspect of programming, because the specifications for any program are continually changing. How many programs can you name that have had only one version? Not many. Well-designed code allows you and others to incorporate prewritten and pretested modules into your program, thus reducing the time to develop a program and yielding code that is more robust and has fewer bugs. One useful feature of the Java programming language is the large supply of prewritten code that you are free to use in your programs.

Programming is an exciting activity. It’s very satisfying to decompose a complex task into computer instructions and watch your program come alive. It can be frustrating, however, when your program either doesn’t run at all or produces the wrong output.

Writing correct programs is critical. Someone’s life or life savings may depend on the correctness of your program. Reusing code helps in developing correct programs, but you must also master effective testing techniques to verify that the output of your program is correct.

In this text, we’ll concentrate not only on the syntax of the Java language, but also on basic programming techniques, good software engineering principles, and effective testing techniques.

Before you can write programs, however, it’s important to understand the platform on which your program will run. A platform refers to the computer hardware and the operating system. Your program will use the hardware for inputting data, for performing calculations, and for outputting results. The operating system will start your program running and will provide your program with essential resources, such as memory, and services, such as reading and writing files.


Motherboard
CPU
with L1 cache
L2 cache
Main memory
Input
(keyboard, LAN)
Output
(monitor, printer, LAN)
Secondary
storage
(hard disk)

1.1 Basic Computer Concepts

1.1.1 Hardware

As shown in **Figure 1.1**, a computer typically includes the following components:

Figure 1.1 A Typical Design of a Personal Computer

Description

- a CPU, or central processing unit, which executes the instructions of a program
- a memory unit, which holds the instructions and data of a program while it is executing
- a hard disk, used to store programs and data so that they can be loaded into memory and accessed by the CPU
- a keyboard used for input of data
- a monitor, used to display output from a program
- an Ethernet port and wireless networking transceiver for connecting to the Internet or a Local Area Network (LAN)

- other components (not shown) such as a graphics card and a DVD drive

For example, if you were to go to a computer store in search of the latest personal computer, you might be shown a computer with this set of specifications:

- a i7-10700KF 4.9GHz with 8 cores Intel processor
- a touch screen
- 8 MB of cache memory
- 32 GB of RAM (Random Access Memory)
- a 1 TB (Terabyte) hard disk

In these specifications, the Intel i7-10700KF is the CPU. Other processors used as CPUs in personal computers and servers include the AMD Ryzen™, the Oracle SPARC, and the IBM POWER processor. When a CPU has multiple cores, for example 8 cores, that means that the CPU can execute several instructions at the same time, running them in parallel.

CPUs consist of an Arithmetic Logic Unit (ALU) [also called an Integer Unit (IU)], which performs basic integer arithmetic and logical operations; a Floating

Point Unit (FPU), which performs floating-point arithmetic; a set of hardware registers for holding data and memory addresses; and other supporting hardware, including a control unit to sequence the instructions. Each CPU comes with its own set of instructions, which are the operations that it can perform. The instructions typically perform arithmetic and logic operations, move data from one location to another, and change the flow of the program (that is, determine which instruction is to be executed next).

A program consists of many instructions. The first step in executing a program is loading it into memory. Each instruction in the program is executed using a **Fetch-Decode-Execute Cycle**. The **Program Counter** keeps track of the next instruction to be fetched. The CPU fetches the next instruction from memory and places it into the **Instruction Register**. The instruction is decoded (is the instruction a move, load, store, etc.?). Then, the instruction is executed. This Fetch-Decode-Execute Cycle repeats until the program ends.

The speed of a CPU is related to its clock cycle, typically rated in GHz (gigahertz); at the time of this edition, a high-end CPU speed would be rated at 5 GHz. The clock speed, measured in clock cycles per second, determines how fast a processor can perform operations such as fetching, decoding, and executing instructions. A CPU rated at 4 GHz executes 4 billion clock cycles a second. Current processors feature pipelining, which allows the CPU to process several instructions at once, so that while one instruction is executing, the processor can decode the next instruction and fetch the next instruction after that. This greatly improves performance of applications.

Memory or storage devices, such as L2 cache, memory, or hard disks, are typically rated in terms of their capacity, expressed in bytes. A **byte** is eight binary digits, or **bits**. A single bit's value is 0 or 1. Depending on the type of memory or storage device, the capacity will be stated in kilobytes, megabytes, gigabytes, or even terabytes. The sizes of these units are shown in **Table 1.1**.

Memory Units and Their Sizes

Memory Unit	**Size**
KB, or Kbytes, or kilobytes	About 1,000 bytes (exactly 2^{10} or 1,024 bytes)
MB, or Mbytes, or megabytes	About 1 million bytes (exactly 2^{20} or 1,048,576 bytes)
GB, or Gbytes, or gigabytes	About 1 billion bytes (exactly 2^{30} or 1,073,741,824 bytes)
TB, or Tbytes, or terabytes	About 1 trillion bytes (exactly 2^{40} or $1.09951\ 3\ 10^{12}$ bytes)

Description

For the CPU to execute at its rated speed, however, instructions and data must be available to the CPU at that speed as well. Instructions and data come directly from the L1 cache, which is memory directly located on the CPU chip. Since the L1 cache is located on the CPU chip, it runs at the same speed as the CPU. However, the L1 cache, which can be several Mbytes, is typically much smaller than main memory, and eventually the CPU will need to process more instructions and data than can be held in the L1 cache at one time.

At that point, the CPU typically brings data from what is called the L2 cache, which is located on separate memory chips connected to the CPU. A typical speed for the L2 cache would be a few nanoseconds (billionths of a second) access time, and this will considerably slow down the rate at which the CPU can execute instructions. L2 cache size today is typically 3 to 8 Mbytes, and again, the CPU will eventually need more space for instructions and data than the L2 cache can hold at one time.

At that point, the CPU will bring data and instructions from main memory, also located outside, but connected to, the CPU chip. This will slow down the CPU even more, because main memory typically has an access time of about 10 to 50 nanoseconds. Main memory, though, is significantly larger in size than the L1 and L2 caches, typically anywhere between 4 and 16 Gbytes. When the CPU runs out of space again, it will have to get its data from the hard disk, which is typically 1 Tbyte or more, but with an access time in the milliseconds (thousandths of a second) range.

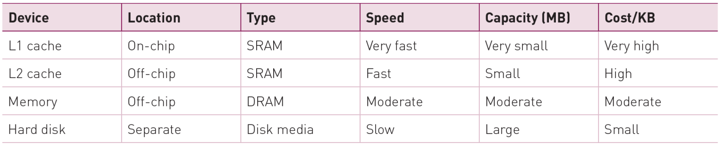
As you can see from these numbers, a considerable amount of speed is lost when the CPU goes from main memory to disk, which is why having sufficient memory is very important for the overall performance of applications.

Another factor that should be taken into consideration is cost per kilobyte. Typically the cost per kilobyte decreases significantly stepping down from L1 cache to hard disk, so high performance is often traded for low price.

Main memory (also called RAM) uses DRAM, or Dynamic Random Access Memory technology, which maintains data only when power is applied to the memory and needs to be refreshed regularly in order to retain data. The L1 and L2 caches use SRAM, or Static Random Access Memory technology, which also needs power but does not need to be refreshed in order to retain data. Memory capacities are typically stated in powers of 2. For instance, 256 Kbytes of memory is 2^{18} bytes, or 262,144 bytes.

Memory chips contain cells, each cell containing a bit, which can store either a 0 or a 1. Cells can be accessed individually or as a group of typically 4, 8, or 16 cells. For instance, a 32-Kbit RAM chip organized as 8K 3 4 is composed of exactly 2^{13}, or 8,192 units, each unit containing four cells. This RAM chip will have four data output pins (or lines) and 13 access pins (or lines), enabling access to all 8,192 units because each access pin can have a value of 0 or 1. **Table 1.2** compares the features of various memory types.

A Comparison of Memory Types



Device	Location	Type	Speed	Capacity (MB)	Cost/KB
L1 cache	On-chip	SRAM	Very fast	Very small	Very high
L2 cache	Off-chip	SRAM	Fast	Small	High
Memory	Off-chip	DRAM	Moderate	Moderate	Moderate
Hard disk	Separate	Disk media	Slow	Large	Small

Description

1.1.2 Operating Systems

An operating system (OS) is a software program that

- controls the peripheral devices (for instance, it manages the file system)
- supports multitasking, by scheduling multiple programs or tasks to execute during the same interval
- allocates memory to each program, so that there is no conflict among the memory of any programs running at the same time
- prevents the user from damaging the system. For instance, it prevents user programs from overwriting the operating system or another program’s memory

The operating system loads, or **boots**, when the computer system is turned on and is intended to run as long as the computer is running.

Examples of operating systems are macOS, Microsoft Windows, Linux, and iOS and Android for mobile devices.

1.1.3 Application Software

Application software consists of the programs written to perform specific tasks. These programs are run by the operating system, or as is typically said, they are run “on top of” the operating system. Examples of applications are word processors, such as Microsoft Word or Apache OpenOffice; spreadsheets, such as Microsoft Excel; database management systems, such as Oracle or Microsoft SQL Server; Internet browsers, such as Mozilla Firefox, Google Chrome, or Microsoft Edge; and most of the programs you will write during your study of computer science.

1.1.4 Computer Networks and the Internet

Computer Networks

Computer networks connect two or more computers. A common network used by many corporations and universities is a LAN, or Local Area Network. A typical LAN connects several computers that are geographically close to one another, often in the same building, and allows them to share resources, such as a printer, a database, or a file system. In a LAN, most user computers are called **clients**, and one or more computers act as **servers**. The server controls access to resources on the network and can supply services to the clients, such as answering database requests, storing and serving files, or managing email. The computers on the network can exchange data through physical cables or through a wireless network.

The Internet

The Internet is a network of networks, connecting millions of computers around the world. The Internet evolved from ARPANET, a 1969 U.S. military research project whose goal was to design a method for computers to communicate. Most computers on the Internet are clients, typically requesting resources, such as webpages, through an Internet browser. These resources are provided by web servers, which store webpages and respond to these requests.

Every machine on the Internet has a unique ID called its IP address (IP stands for Internet Protocol). Special computers called **routers** find a path through the Internet from one computer to another using these IP addresses. A computer can have a static IP address, which is dedicated to that machine, or a dynamic IP address, which is assigned to the computer when it connects to the Internet. An IP address is made up of four octets, whose values in decimal notation are between 0 and 255. For instance, 58.203.151.103 could represent such an IP address. In binary notation, this IP address is 111010.11001011.10010111.1100111. Later in this chapter, we will learn how to convert a decimal number, such as 103, to its binary equivalent, 1100111.

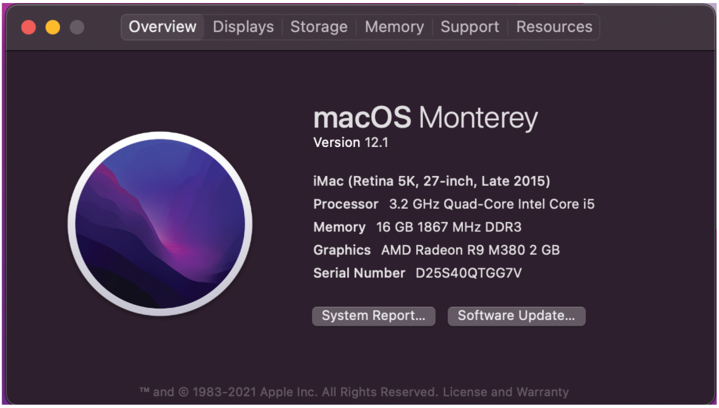
Most people are familiar with URL (Uniform Resource Locator) addresses that look like *http://www.oracle.com/technetwork/java/index.html.* A URL is actually an Internet domain name as well as the path on that domain to a specific web page. In this URL, the domain name is *www.oracle.com.* The page requested is *index.html,* which is located in the *java* folder under the *technetwork* folder.

Domain name resolution servers, which implement the Domain Name System (DNS), convert domain names to IP addresses, so that Internet users don't need to know the IP addresses of websites they want to visit. The World Wide Web Consortium (W3C), an international group developing standards for Internet access, prefers the term Uniform Resource Identifier (URI) rather than URL, because URI covers future Internet addressing schemes.

with these end-of-chapter questions

1.7.1	Multiple Choice Exercises Questions 1, 2, 3, 4
1.7.3	General Questions Questions 21, 22, 23
1.7.4	Technical Writing Questions 31, 32, 33

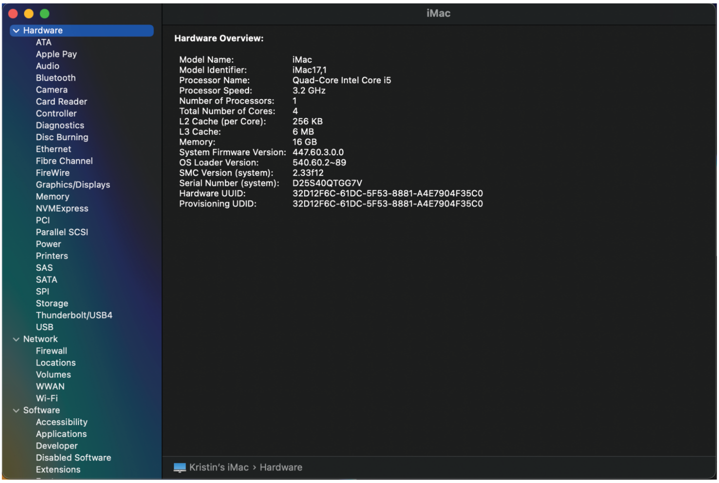
Item	Value
OS Name	Microsoft Windows 11 Home
Version	10.0.22000 Build 22000
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	DESKTOP-CVBK8EQ
System Manufacturer	Dell Inc.
System Model	Inspiron 5515
System Type	x64-based PC
System SKU	0A78
Processor	AMD Ryzen 7 5700U with Radeon Graphics, 1801 Mhz, 8 Core(s), 16 Logical...
BIOS Version/Date	Dell Inc. 1.4.0, 7/12/2021
SMBIOS Version	3.3
Embedded Controller Version	255.255
BIOS Mode	UEFI
BaseBoard Manufacturer	Dell Inc.
BaseBoard Product	0KDKG8
BaseBoard Version	A00
Platform Role	Mobile
Secure Boot State	On
PCR7 Configuration	Elevation Required to View
Windows Directory	C:\Windows
System Directory	C:\Windows\system32
Boot Device	\Device\HarddiskVolume1
Locale	United States
Hardware Abstraction Layer	Version = "10.0.22000.1"
User Name	DESKTOP-CVBK8EQ\herve
Time Zone	Eastern Standard Time
Installed Physical Memory (RA...	16.0 GB
Total Physical Memory	15.4 GB
Available Physical Memory	7.58 GB
Total Virtual Memory	17.7 GB
Available Virtual Memory	5.90 GB
Page File Space	2.38 GB
Page File	C:\pagefile.sys


Overview
Displays
Storage
Memory
Support
Resources
System Report...
Software Update...

macOS Monterey
Version 12.1

iMac (Retina 5K, 27-inch, Late 2015)
Processor 3.2 GHz Quad-Core Intel Core i5
Memory 16 GB 1867 MHz DDR3
Graphics AMD Radeon R9 M380 2 GB
Serial Number D25S40QTGG7V

™ and © 1983-2021 Apple Inc. All Rights Reserved. License and Warranty


Hardware
ATA
Apple Pay
Audio
Bluetooth
Camera
Card Reader
Controller
Diagnostics
Disc Burning
Ethernet
Fibre Channel
FireWire
Graphics/Displays
Memory
NVMExpress
PCI
Parallel SCSI
Power
Printers
SAS
SATA
SPI
Storage
Thunderbolt/USB4
USB
Network
Firewall
Locations
Volumes
WWAN
Wi-Fi
Software
Accessibility
Applications
Developer
Disabled Software
Extensions
iMac
Kristin's iMac › Hardware

```
Hardware Overview:

  Model Name:                 iMac
  Model Identifier:           iMac17,1
  Processor Name:             Quad-Core Intel Core i5
  Processor Speed:            3.2 GHz
  Number of Processors:       1
  Total Number of Cores:      4
  L2 Cache (per Core):        256 KB
  L3 Cache:                   6 MB
  Memory:                     16 GB
  System Firmware Version:    447.60.3.0.0
  OS Loader Version:          540.60.2~89
  SMC Version (system):       2.33f12
  Serial Number (system):     D25S40QTGG7V
  Hardware UUID:              32D12F6C-61DC-5F53-8881-A4E7904F35C0
  Provisioning UDID:          32D12F6C-61DC-5F53-8881-A4E7904F35C0
```

1.2 Practice Activity: Displaying System Configuration

We have explored hardware and operating systems in general. Now, let's discover some information about the hardware and operating system on your computer. Depending on whether you're using a Windows operating system or a Mac OS operating system, choose the appropriate directions that follow to display the operating system's name, the CPU type, and how much memory the computer has.

1.2.1 Displaying Windows Configuration Information

To display system configuration information on a computer running Windows 11, run *msinfo32.exe* from the command line. From the *Start* menu, type *cmd* to start the Command Prompt program and then type *msinfo32* into the Command Prompt window. You will get a display similar to the one in **Figure 1.2**, although the information displayed varies, depending on your hardware and the version of Windows you are running.

Figure 1.2 Windows System Information

Screen shots reprinted with permission from Microsoft.

Description

As you can see in Figure 1.2, this computer is running AMD Ryzen 7 5700U CPU processor with 8 cores running at 1801MHz, and the computer has 16 Gbytes of memory, 7.58 Gbytes of which are not being used at the time of the display.

1.2.2 Displaying Mac OS Configuration Information

To display system information on a Mac OS computer, click on the apple icon on the top left of the screen and select “About This Mac.” You will see something similar to the display in **Figure 1.3**, which displays the version of the MAC OS operating system that is running on the computer.

Figure 1.3 Mac Operating System Information

Screen shots reprinted with permission from Apple Inc.

Description

As you can see, this iMac is running macOS Monterey Version 12.1. The processor is an Intel Core i5 running at 3.2 GHz and the iMac has 16 Gbytes of memory. Your information may be different from that shown in the figure, depending on your hardware and operating system version.

DISCUSSION QUESTIONS

1. Compare the system information on several computers. Is it the same or different from computer to computer? Explain why the information is the same or different.
2. In the sample display for Windows 11, the computer has 16 Gbytes of memory, but only 7.58 Gbytes of memory is available. Why do you think some memory is not available?
3. Compare your computer to the ones on the previous pages. Which do you think would have better performance? Explain your answer.

1.3 Data Representation

1.3.1 Binary Numbers

As mentioned earlier, a CPU understands only binary numbers, whose digits consist of either 0 or 1. All data is stored in a computer's memory as binary digits. A bit holds one binary digit. A byte holds eight binary digits.

Binary numbers are expressed in the base 2 system, because there are only 2 values in that system, 0 and 1. By contrast, most people are used to the decimal, or base 10, system, which uses the values 0 through 9.

There are other number systems, such as the octal, or base 8, system, which uses the digits from 0 to 7, and the hexadecimal, or base 16, system, which uses the digits 0 to 9 and the letters A to F.

As we know it in the decimal system, the number 359 is composed of the following three digits:

3, representing the hundreds, or 10^2
5, representing the tens, or 10^1
9, representing the ones, or 10^0

Therefore, we can write 359 as

$$359 = 3*10^2 + 5*10^1 + 9*10^0$$

Description

Thus, the decimal number 359 is written as a linear combination of powers of 10 with coefficients from the base 10 alphabet, that is, the digits from 0 to 9. Similarly, the binary number 11011 is written as a linear combination of powers of 2 with coefficients from the base 2 alphabet, that is, the digits 0 and 1.

For example, the binary number 11011 can be written as

$$11011 = 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$$

Description

Table 1.3 lists the binary equivalents for the decimal numbers 0 through 8, while **Table 1.4** lists the decimal equivalents of the first 15 powers of 2.

Binary Equivalents of Decimal Numbers 0 Through 8

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000

Description

Powers of 2 and Their Decimal Equivalents

2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
16,384	8,192	4,096	2,048	1,024	512	256	128	64	32	16	8	4	2	1

Description

Note that in Table 1.3, as we count in increments of 1, the last digit alternates between 0 and 1. In fact, we can see that for even numbers, the last digit is always 0 and for odd numbers, the last digit is always 1.

Because computers store numbers as binary, and people recognize numbers as decimal values, conversion between the decimal and binary number systems often takes place inside a computer.

Let's try a few conversions. To convert a binary number to a decimal number, multiply each digit in the binary number by $2^{position21}$, counting the rightmost position as position 1 and moving left through the binary number. Then add the products together.

Using this method, let's calculate the equivalent of the binary number 11010 in our decimal system.

$$\begin{aligned} 11010 &= 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 \\ &= 16 + 8 + 0 + 2 + 0 \\ &= 26 \end{aligned}$$

Description

Now let's examine how to convert a decimal number to a binary number. Let's convert the decimal number 359 into its binary number equivalent. As we can see from the way we rewrote 11011, a binary number can be written as a sum of powers of 2 with coefficients 0 and 1.

The strategy to decompose a decimal number into a sum of powers of 2 is simple: first find the largest power of 2 that is smaller than or equal to the decimal number, subtract that number from the decimal number, then do the same with the remainder, and so on, until you reach 0.

The largest power of 2 that is smaller than 359 is 256, or 2^8 (the next larger power of 2 would be 512, which is larger than

359). Subtracting 256 from 359 gives us 103 (359 2 256 5 103), so we now have

$$359 = 2^8*1 + 103$$

Description

Now we apply the same procedure to 103. The largest power of 2 that is smaller than 103 is 64, or 2^6. That means that there is no factor for 2^7, so that digit’s value is 0. Subtracting 64 from 103 gives us 39.

Now we have

$$359 = 2^8*1 + 2^7*0 + 2^6*1 + 39$$

Description

Repeating the procedure for 39, we find that the largest power of 2 smaller than 39 is 32 or 2^5. Subtracting 32 from 39 gives us 7.

So we now have

$$359 = 2^8*1 + 2^7*0 + 2^6*1 + 2^5*1 + 7$$

Description

Repeating the procedure for 7, the largest power of 2 smaller than 7 is 2^2, or 4. That means that there are no factors for 2^4 or 2^3, so the value for each of those digits is 0. Subtracting 4 from 7 gives us 3, so we have

$$359 = 2^8*1 + 2^7*0 + 2^6*1 + 2^5*1 + 2^4*0 + 2^3*0 + 2^2*1 + 3$$

Description

Repeating the procedure for 3, the largest power of 2 smaller than 3 is 2, or 2^1, and we have:

$$359 = 2^8*1 + 2^7*0 + 2^6*1 + 2^5*1 + 2^4*0 + 2^3*0 + 2^2*1 + 2^1*1 + 1$$

Description

1 is a power of 2; it is 2^0, so we finally have

$$359 = 2^8*1 + 2^7*0 + 2^6*1 + 2^5*1 + 2^4*0 + 2^3*0 + 2^2*1 + 2^1*1 + 2^0*1$$

Description

Removing the power of 2 multipliers, 359 can be represented in the binary system as

$$\begin{array}{lllllllllllllllllll} 359 & = & 2^8*1 & + & 2^7*0 & + & 2^6*1 & + & 2^5*1 & + & 2^4*0 & + & 2^3*0 & + & 2^2*1 & + & 2^1*1 & + & 2^0*1 \\ & = & 1 & & 0 & & 1 & & 1 & & 0 & & 0 & & 1 & & 1 & & 1 \end{array}$$

Description

or

$$1\ 0110\ 0111$$

CODE IN ACTION

To see a step-by-step demonstration of converting between decimal and binary numbers, look for the movie within the online resources. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

In a computer program, we will use both positive and negative numbers. Appendix D explains how negative numbers, such as 234, are represented in the binary system. In a computer program, we also use floating-point numbers, such as 3.75. Appendix E explains how floating-point numbers are represented using the binary system.

1.3.2 Using Hexadecimal Numbers to Represent Binary Numbers

As you can see, binary numbers can become rather long. With only two possible values, 0 and 1, it takes 16 binary digits to represent the decimal value 132,768. For that reason, the hexadecimal, or base 16, system is often used as a shorthand representation of binary numbers. The hexadecimal system uses 16 digits: 0 to 9 and A to F. The letters A to F represent the values 10, 11, 12, 13, 14, and 15.

The maximum value that can be represented in four binary digits is 2^4 2 1, or 15. The maximum value of a hexadecimal digit is also 15, which is represented by the letter F. So you can reduce the size of a binary number by using hexadecimal digits to represent each group of four binary digits.

Table 1.5 displays the hexadecimal digits along with their binary equivalents.

Hexadecimal Digits and Equivalent Binary Values

Hex Digit	Binary Value	Hex Digit	Binary Value
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Description

To represent the following binary number in hexadecimal, you simply substitute the appropriate hex digit for each set of four binary digits.

```
0001 1010 1111 1001 1011 0011 1011 1110
  1    A    F    9    B    3    B    E
```

Description

Here’s an interesting sequence of hexadecimal numbers. The first 32 bits of every Java applet are:

```
1100 1010 1111 1110 1011 1010 1011 1110
```

Description

Translated into hexadecimal, that binary number becomes:

CAFE BABE

Description

1.3.3 Representing Characters with the Unicode Character Set

Java represents characters using the Unicode Worldwide Character Standard, or simply Unicode. Each Unicode character is represented as 16 bits, or two bytes. This means that the Unicode character set can encode 65,536 characters.

The Unicode character set was developed by the Unicode Consortium, which consists of computer manufacturers, software vendors, the governments of several nations, and others. The consortium’s goal was to support an international character set, including the printable characters on the standard QWERTY keyboard, as well as international characters such as e or λ.

Many programming languages store characters using the ASCII (American Standard Code for Information Interchange) character set, which uses 7 bits to encode each character, and thus, can represent only 128 characters. For compatibility with the ASCII character set, the first 128 characters in the Unicode character set are the same as the ASCII character set.

Table 1.6 shows a few examples of Unicode characters and their decimal equivalents.

Selected Unicode Characters and Their Decimal Equivalents

Unicode Character	Decimal Value	Hex Value
NUL, the null character (a nonprintable character)	0	0000
*	42	002A
1	49	0031
2	50	0032

A	65	0041
B	66	0042
a	97	0061
b	98	0062
}	125	007D
delete (a nonprintable character)	127	007F

Description

For more information on the Unicode character set, see Appendix C or visit the Unicode Consortium's website at *http://www.Unicode.org*.

with these end-of-chapter questions

1.7.1	Multiple Choice Exercises Questions 5, 6, 7, 8
1.7.2	Converting Numbers Questions 15, 16, 17, 18, 19, 20
1.7.3	General Questions Questions 24, 25, 26

1.4 Programming Languages

1.4.1 High- and Low-Level Languages

Programming languages can be categorized into three types:

- machine language
- assembly language
- high-level language

In the early days of computing, programmers often used machine language or assembly language. Machine language uses binary codes, or strings of 0s and 1s, to execute the instruction set of the CPU and to refer to memory addresses. This method of programming is extremely challenging and time consuming. Also, the code written in machine language is not portable to other computer architectures because each CPU has its own set of instructions. Machine language's early popularity can be attributed largely to the fact that programmers had no other choices. However, programmers rarely use machine language today.

Assembly languages are one step above machine language, using symbolic names for memory addresses and mnemonics for processor instructions—for example: *BEQ* (branch if equal), *SW* (store), or *LW* (load). An Assembler program converts the code to machine language before it is executed. Like machine language, assembly languages are also CPU-dependent and are not portable among computers with different processors (for instance, between Intel and SPARC). Assembly language is easier to write than machine language but still requires a significant effort and, thus, is usually used only when the program requires features, such as direct hardware access, that are not supported by a high-level language.

High-level languages, such as Java, C+ + , Swift, PHP, and Python, are closer to the English language than they are to machine language, making them a lot easier to use for software development and more portable among CPU architectures. For this reason, programmers have embraced high-level languages for more and more applications.

Characteristics of high-level languages, such as Java, are

- The languages are highly symbolic. Programmers write instructions using keywords and special characters and use symbolic names for data.
- The languages are somewhat portable (some more portable than others) among different CPUs.
- Some high-level programming languages, such as C++ , Java, and Python, can be used for general-purpose applications, and some high-level programming languages can be used for specialized applications, such as Swift for iPhone development and PHP for Internet applications.

High-level languages are compiled, interpreted, or a combination of both. A program written in a compiled language, such as C++ , is converted by a compiler into machine code, then the machine code is executed.

By contrast, a program written using an interpreted language, such as Perl, is read and converted to machine code, line by line, at execution time. Typically, a program written in an interpreted language will run more slowly than its equivalent written in a compiled language.

Java uses a combination of a compiler and an interpreter. A Java program is first compiled into processor-independent **byte codes**, then the byte code file is interpreted at run time by software called the **Java Virtual Machine (JVM)**.

1.4.2 An Introduction to Object-Oriented Programming

Initial high-level languages, such as Fortran or C, were procedural. Typically, programmers wrote task-specific code in separate procedures, or functions, and invoked these procedures from other sections of the program in order to perform various tasks. The program’s data was generally shared among the procedures.

In the mid-1970s, the first object-oriented programming language, Smalltalk, was introduced, enabling programmers to write code with a different approach. Whereas procedures or functions dealt mainly with basic data types such as integers, real numbers, or single characters, Smalltalk provided the programmer with a new tool: classes and objects of those classes.

A class enables the programmer to encapsulate data and the functions to manipulate that data into one package. A class essentially defines a template, or model, from which objects are created. Creating an object is called **instantiation**. Thus, objects are created—instantiated—according to the design of the class.

A class could represent something in real life, such as a person. The class could have various attributes such as, in the example of a "person" class, a first name, a last name, and an age. The class would also provide code, called **methods**, that allow the creator of the object to set and retrieve the values of the attributes.

One big advantage to object-oriented programming is that well-written classes can be reused in new programs, thereby reducing future development time.

Smalltalk was somewhat successful, but had a major deficiency: its syntax was unlike any syntax already known by most programmers. Most programmers who knew the procedural C programming language were attracted by the object-oriented features of Smalltalk but were reluctant to use it because its syntax was so different from C's syntax. C+ + added object-oriented features to C, but also added complexity.

Meanwhile, the Internet was growing by leaps and bounds and gaining popularity daily. Web developers used HTML to develop webpages and soon felt the need to incorporate programming features not only on the server side, but also directly on the client side. Fortunately, Java appeared on the scene.

1.4.3 The Java Language

On May 23, 1995, Sun Microsystems introduced Java, originally named Oak, as a free, object-oriented language targeted at embedded applications for consumer devices. A Java Virtual Machine was incorporated immediately into the Netscape

Navigator Internet browser, and as the Internet grew, small Java programs, known as applets, began to appear on webpages in increasing numbers. Java syntax is basically identical (with some minor exceptions) to that of C+ +, and soon programmers all over the world started to realize the benefits of using Java. Those benefits include

- syntax identical to that of C + +, except that Java eliminates some of C ++ “s more complex features
- object orientation
- Internet-related features, such as servlets, which are run by the web server
- an extensive library of classes that can be reused readily, including classes for providing a Graphical User Interface and Java Database Connectivity (JDBC) for communicating with a database
- portability among every platform that supports a Java Virtual Machine
- built-in networking
- open source availability of the Java Development Kit

As we mentioned earlier, a Java program is first compiled into processorindependent byte codes, then the byte codes are interpreted at run time by the Java Virtual Machine (JVM). As its name implies, the JVM simulates a virtual processor with its own instruction set, registers, and instruction pointer. Thus, to run a Java program, we only need a JVM. Fortunately, JVMs are available on every major computing platform.

Because Java programs are interpreted at run time, they typically run more slowly than their C++ counterparts. However, many platforms provide Java compilers that convert source code directly to machine code. This results in greater execution speed, but with an accompanying loss of portability. Just-inTime (JIT) compilers are also available. These JITs compile code at run time so that subsequent execution of the same code runs much faster.

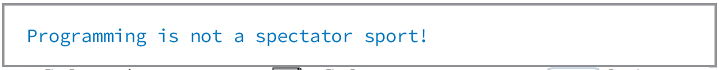
Java programs can be written as servlets, Android apps, or applications.

Java servlets are invoked by the web server and run on the server, without being downloaded to the client. Typically, servlets dynamically generate web content by reading and writing to a database using JDBC (Java Database Connectivity).

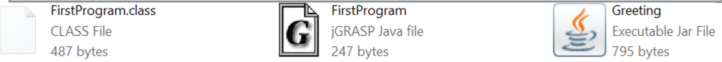
The Android Open Source project has chosen Java to be the language for developing apps for Android devices, such as cell phones, tablets, and smart TVs.

Java applications run standalone on a client computer. In this text, we will write Java applications.

Oracle Corporation, which acquired Sun Microsystems in January 2010, provides a valuable Java website (*www.oracle.com/technetwork/java*), which has information on using the prewritten classes, a tutorial on Java, and many more resources for the Java programmer. We will refer you to that site often in this text.



```
Programming is not a spectator sport!
```



FirstProgram.class
CLASS File
487 bytes

FirstProgram
jGRASP Java file
247 bytes

Greeting
Executable Jar File
795 bytes

1.5 An Introduction to Programming

1.5.1 Programming Basics

In many ways, programming is like solving a puzzle. We have a task to perform and we know the operations that a computer can perform (input, calculations, comparisons, rearranging of items, and output). As programmers, our job is to decompose a task into individual, ordered steps of inputting, calculating, comparing, rearranging, and outputting.

For example, suppose our task is to find the sum of two numbers. First, our program needs to read (input) the numbers into the computer. Next, our program needs to add the two numbers together (calculate). Finally, our program needs to write (output) the sum.

Notice that this program consists of steps, called **instructions**, which are performed in order (“First,” “Next,” “Finally”). Performing operations in order, one after another, is called **sequential processing**.

The order in which instructions are executed by the computer is critical in programming. You can’t calculate the sum of two numbers before you have read the two numbers, and you can’t output a sum before you have calculated it. Programming, therefore, requires the programmer to specify the ordering of instructions, which is called the **flow of control** of the program. There are four different ways that the flow of control can progress through a program: sequential execution, method call, selection, and looping. We’ve just seen sequential execution, and we’ll discuss the other types of flow of control in the next section.

Because getting the flow of control correct is essential to getting a program to produce correct output, programmers use a tool called **pseudocode** (pronounced *sue dough code*) to help them design the flow of control before writing the code.

1.5.2 Program Design with Pseudocode

Pseudocode, from *pseudo*, which means “appearing like,” is a method for expressing a program’s order of instructions in the English language, rather than a programming language. In this way, the programmer can concentrate on designing a program without also being bogged down in the syntax of the particular programming language.

The pseudocode for calculating the sum of two numbers would look like **Example 1.1**.

Example 1.1 Pseudocode for Summing Two Numbers

```
read first number
read second number
set total to (first number + second number)
output total
```

Description

Fortunately, the rules for writing pseudocode are not rigid. Essentially, you can use any wording that works for you.

Let’s look at another example. Suppose our program needs to calculate the square root of a number. The instructions for calculating a square root are rather complex; fortunately, Java provides prewritten code that computes the square root of a number. The prewritten code is called a **method**, and our program can execute that code by **calling the method**. As part of the method call, we tell the method which number’s square root we want to calculate. This is called **passing an argument to the method**. When the method finishes executing its instructions, control is passed back to our program just after the method call and our program can use the square root of the number calculated by the method. This is called **returning a value**. Another way of looking at method calls is to consider what happens when we’re reading a book and find a word we don’t understand. We mark our place in the book and look up the word in a dictionary. When we’re finished looking up the word, we go back to our place in the book and continue reading.

Example 1.2 shows the pseudocode for calculating the square root of an integer.

Example 1.2 Using a Method Call to Calculate a Square Root

```
read a number
call the square root method, passing the number and
   receiving the square root
output the square root of the number
```

Description

The order of operations is still input, calculate, and output, but we're calling a method to perform the calculation for us.

Now suppose our task is to determine whether a number is positive or negative. First, our program should input the number into the computer. Next, we need to determine whether the number is positive or negative. We know that numbers greater than or equal to 0 are positive and numbers less than 0 are negative, so our program should compare the number to 0. Finally, our program should write a message indicating whether the number is positive or negative.

Like Examples 1.1 and 1.2, the operations are input, calculate, and output, in that order. However, depending on whether the number is positive or negative, our program should write a different message. If the number is greater than or equal to 0, the program should write a message that the number is positive, but if the number is less than 0, the program should write a message that the number is negative. Code used to handle this situation is called **selection**; the program selects which code to execute based on the value of the data.

The pseudocode for this program could be written as that shown in **Example 1.3**.

Example 1.3 Using Selection

```
read a number
if the number is greater than or equal to 0
  write "Number is positive."
else
  write "Number is negative."
```

Description

Notice the indentation for the code that will be selected based on the comparison of the number with 0. Programmers use indentation to make it easier to see the flow of control of the program.

Now let's get a little more complicated. Suppose our program needs to find the sum of a group of numbers. This is called **accumulating**. To accomplish this, we can take the same approach as if we were adding a group of numbers using a calculator. We start with a total of 0 and add each number, one at a time, to the running total. When we have no more numbers to add, the running total is the total of all the numbers.

Translating this into pseudocode, we get the code shown in **Example 1.4**.

Example 1.4 Accumulating a Total

```
set total to 0
read a number
while there was a number to read, repeat next two instructions
  add number to total
  read the next number
write total
```

Description

The indented code will be repeated for each number read until there are no more numbers. This repeated execution of the same code is called **looping**, or **iteration**, and is used extensively in programming whenever the same processing needs to be performed on each item in a set.

Accumulating a total and determining whether a number is positive or negative are just two of many commonly performed operations. In programming, we will often perform tasks for which there are standard methods of processing, called **algorithms**. For example, the algorithm for accumulation is to set a total to 0, use looping to add each item to the total, then output the total. More generally, we can think of an algorithm as a strategy to solve a problem. Earlier in the chapter, we used an algorithm to convert a decimal number to its binary representation.

Other common programming tasks are counting items, calculating an average, sorting items into order, and finding the minimum and maximum values. In this text, we will learn the standard algorithms for performing these common operations. Once we learn these algorithms, our programming job will become easier. When we recognize that a program requires any of these tasks, we can simply plug in the appropriate algorithm with some minor modifications.

Programming, in large part, is simply reducing a complex task to a set of subtasks that can be implemented by combining standard algorithms that use sequential processing, method calls, selection, and looping.

The most difficult part of programming, however, is recognizing which algorithms to apply to the problem at hand. This requires analytical skills and the ability to see patterns. Throughout this text, we will point out common patterns wherever possible.

SOFTWARE ENGINEERING TIP

Looking for patterns will help you determine the appropriate algorithms for your programs.

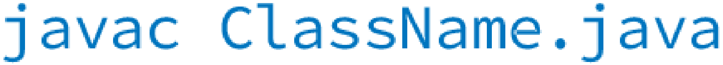
1.5.3 Developing a Java Application

Writing a Java application consists of several steps: writing the code, compiling the code, and executing the application. Java source code is stored in a text file with the extension *. java*. Compiling the code creates one or more *. class* files, which contain processor-independent byte codes. The Java Virtual Machine (JVM) translates the byte codes into machine-level instructions for the processor on which the Java application is running. Thus, if a Java application is running on an Intel i7 processor, the JVM translates the byte codes into the i7’s instruction set.

Oracle provides a Java SE Development Toolkit (JDK) on its website (*www .oracle.com/technetwork/java),* which is downloadable free of charge. The JDK contains a compiler, JVM, and jshell, which is a tool for immediate execution of Java code. In addition, the JDK contains a broad range of prewritten Java classes that programmers can use in their Java applications.

If you are downloading and installing Java yourself, be sure to follow the directions on the Oracle website, including the directions for setting the path for *javac,* the Java compiler. You need to set the path correctly so that you can run the Java compiler from any folder on your computer.

To develop an application using the JDK, write the source code using any text editor, such as Notepad on Windows, TextEdit on Macs, or the vi editor on Linux. To compile the code, invoke the compiler from the command line:



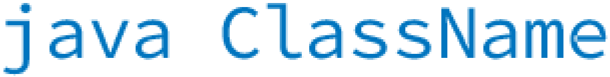
```
javac ClassName.java
```

Description

where *ClassName.java* is the name of the source file.

If our program, written in the file *ClassName.java,* compiles correctly, a new file, *ClassName.class,* will be created in our current directory.

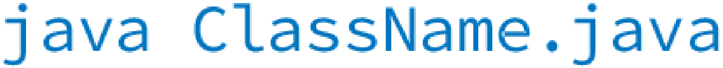
To run the application, we invoke the JVM from the command line:



```
java ClassName
```

Description

Since Java version 11, we can now compile and run a program using the *java* command only, without using *javac*, as in:

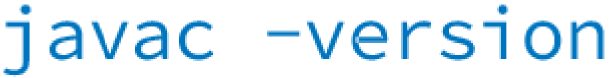


```
java ClassName.java
```

Description

Note that if we do that, the *ClassName.class* file will not be generated.

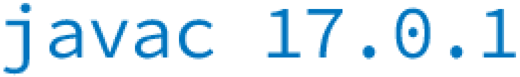
We can check what version of *javac* we have installed by typing:



```
javac -version
```

Description

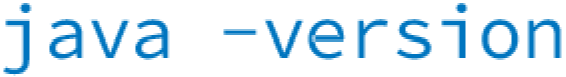
An example of output is:



```
javac 17.0.1
```

Description

In order to check what version of *java* we have installed, we type:



```
java -version
```

Description

An example of output is:

```
java version "17.0.1" 2021-10-19 LTS
Java(TM) SE Runtime Environment (build 17.0.1+12-LTS-39)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.1+12-LTS-39, mixed mode, sharing)
```

Description

Typically, programmers use an Integrated Development Environment (IDE) to develop applications rather than invoking the compiler and JVM from the command line. An IDE consists of a program editor, a compiler, and a run-time environment, integrated via a Graphical User Interface. The advantage to using an IDE is that errors in the Java code that are found by the compiler or the JVM can be linked directly to the program editor at the line in the source file that caused the error. Additionally, the Graphical User Interface enables the programmer to switch among the editor, compiler, and execution of the program without launching separate applications.

Some of the many available IDEs include Eclipse from the Eclipse Foundation, Inc.; IntelliJ from JetBrains s.r.o.; JGRASP, developed at Auburn University; NetBeans, downloadable from Oracle; and TextPad from Helios Software Solutions. Some IDEs are freely available, while others require a software license fee.

with these end-of-chapter questions

1.7.1	Multiple Choice Exercises Questions 9, 10, 11, 12, 13, 14
1.7.3	General Questions Questions 27, 28, 29, 30
1.7.4	Technical Writing Question 34

1.5.4 Programming Activity 1: Writing a First Java Application

Let’s create our first Java program. This program prints the message, “Programming is not a spectator sport!”

Start by launching your IDE and open a new editor window. This is where you will write the code for the program.

Before you type any code, however, let’s name the document. You do this by saving the document as *FirstProgram.java*. Be sure to capitalize the F and the P and keep the other letters lowercase. Java is case-sensitive, so Java considers *firstprogram.java* or even *Firstprogram.java* to be a different name. Also, note that there is no space between *First* and *Program*.

Keeping case sensitivity in mind, type in the program shown in **Example 1.5**.

Example 1.5 A First Program in Java

```java
// First program in Java
// Anderson, Franceschi

public class FirstProgram
{
  public static void main( String [ ] args )
  {
    System.out.println( "Programming is not a spectator sport!" );

    System.exit( 0 );
  }
}
```

Description

At this point, we ask that you just type the program as you see it here, except for the line numbers, which are not part of the program. Line numbers are displayed in this example to allow easy reference to a particular line in the code. We'll explain a little about the program now; additional details will become clear as the semester progresses.

COMMON ERROR TRAP

Java is case-sensitive. The class name and the file name must match exactly, including capitalization.

The first two lines, which start with two forward slashes, are comments. They will not be compiled or executed; they are simply information for the programmer and are used to increase the readability of the program.

Line 4 defines the class name as *FirstProgram*. Notice that the class name must be spelled exactly the same way—including capitalization—as the file name, *FirstProgram.java*.

The curly braces in lines 5 and 12 mark the beginning and ending of the *FirstProgram* class, and the curly braces in lines 7 and 11 mark the beginning and ending of the *main* method. Every Java application must define a class and a *main* method. Execution of a Java application always begins with the code inside *main*. So when this application begins, it will execute line 8, which writes the message "*Programming is not a spectator sport!*" to the system console. Next, it executes line 10, *System.exit(0),* which exits the program. Including this line is optional; if you omit this line, the application will still exit normally.

As you type the program, notice that your IDE automatically colors your text to help you distinguish comments, *String* literals ("*Programming is not a spectator sport!*"), Java class names (*String, System*), and keywords (*public, class, static*), which are reserved for specific uses in Java. Curly braces, brackets, and parentheses, which have syntactical meaning in Java, are usually displayed in color as well. Your IDE may use different colors than those shown in Example 1.5.

When you have completed typing the code in Example 1.5, compile it. If everything is typed correctly, the compiler will create a *FirstProgram.class* file, which contains the byte codes for the program.

If you received any compiler errors, check that you have entered the code exactly as it is written in Example 1.5. We give you tips on finding and fixing the errors in the next section.

If you got a clean compile with no errors, congratulations! You're ready to execute the application. This will invoke the JVM and pass it the *First Program .class* file created by the compiler. If all is well, you will see the message, *Programming is not a spectator sport!*, displayed on the Java console, which is the text window that opens automatically. **Figure 1.4** shows the correct output of the program.

Figure 1.4 Output from Example 1.5

Debugging Techniques

If the compiler found syntax errors in the code, these are called **compiler errors**, not because the compiler caused them, but because the compiler found them. When the compiler detects errors in the code, it writes diagnostic information about the errors.

For example, try typing *println* with a capital P (as *Println*), and recompiling. The compiler displays the following message:

```
FirstProgram.java:8: error: cannot find symbol
    System.out.Println( "Programming is not a spectator sport!" );
              ^
  symbol: method Println(String)
  location: variable out of type PrintStream
1 error
```

Description

The first line identifies the file name that contains the Java source code, as well as the line number in the source code where the error occurred. In this case, the error occurred on line 8. The second line displays line 8 from the source code with a caret (^) pointing to *Println* as being the cause of the error. The symbol and location information in the third and fourth lines indicate that the *Println* method is unknown. Remember that Java is case-sensitive, so *println* and *Println* are considered to be different. As you gain experience with Java, these error messages will become more meaningful to you. With most IDEs, double-clicking on the first line in the error message transfers you to the source code window with your cursor positioned on line 8 so you can correct the error.

Many times, the compiler will find more than one error in the source code. When that happens, don’t panic. Often, a single problem, such as a missing semicolon or curly brace, can cause multiple compiler errors.

For example, after correcting the preceding error, try deleting the left curly brace in line 7, then recompiling. The compiler reports four errors:

```
FirstProgram.java:6: error: ';' expected
        public static void main( String [ ] args )
                                                  ^

FirstProgram.java:10: error: <identifier> expected
        System.exit( 0 );
                   ^

FirstProgram.java:10: error: illegal start of type
        System.exit( 0 );
                     ^

FirstProgram.java:12: error: class, interface or enum expected
}
^
4 errors
```

Description

As you can see, the compiler messages do not always report the problem exactly. When you receive a compiler message, looking at the surrounding lines will often help you find the error. Depending on your IDE, you might see messages other than those shown here because some IDEs attempt to interpret the error messages from the compiler to provide more relevant information on the errors.

SOFTWARE ENGINEERING TIP

Because one syntax error can cause multiple compiler errors, correct only the obvious errors and recompile after each correction.

It is sometimes easier to fix one error at a time and recompile after each fix, because the first fix might eliminate many of the reported errors.

When all the compiler errors are corrected, you’re ready to execute the program.

It is possible to get a clean bill of health from the compiler, yet the program still won't run. To demonstrate this, try eliminating the brackets in line 6 after the word *String*. If you then compile the program, no errors are reported. But when you try to run the program, you get a **run-time error**.

Instead of *Programming is not a spectator sport!*, the following message is displayed on the Java console:

```
Error: Main method not found in class FirstProgram, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Description

This means that the *main* method header (line 6) was not typed correctly.

Thus, we've seen that two types of errors can occur while you are developing a Java program: compiler errors, which are usually caused by language syntax errors or misspellings, and run-time errors, which are often caused by problems using the prewritten classes. Run-time errors can also be caused by exceptions that the JVM detects as it is running, such as an attempt to divide by zero.

Testing Techniques

Once your program compiles cleanly and executes without run-time errors, you may be tempted to conclude that your job is finished. Far from it—you must also verify the results, or output, of the program.

In the sample program, it's difficult to get incorrect results—other than misspelling the message or omitting the spaces between the words. But any nontrivial program should be tested thoroughly before declaring it production ready.

To test a program, consider all the possible inputs and the corresponding correct outputs. It often isn't feasible to test every possible input, so programmers usually test **boundary conditions**, which are the values that sit on the boundaries of producing different output for a program.

For example, to test the code that determines whether an integer is negative or nonnegative, you would feed the program -1 and 0. These are the boundaries of negative and nonnegative integers. In other words, the boundary between negative and nonnegative integers is between -1 and 0.

When a program does not produce the correct output, we say the program contains **logic errors**. By testing your program thoroughly, you can discover and correct most logic errors. **Table 1.7** shows types of program errors and their usual causes.

Types of Program Errors and Their Causes

Type of Error	Usual Causes
Compiler errors	Incorrect language syntax or misspellings
Run-time errors	Incorrect use of classes
Logic errors	Incorrect program design or incorrect implementation of the design

Description

We'll talk more about testing techniques throughout the text.

DISCUSSION QUESTIONS

1. In the Debugging Techniques section, we saw that making one typo could generate several compiler errors. Why do you think that happens?
2. Explain why testing boundary conditions is an efficient way to verify a program's correctness.
3. Did any errors occur while you were developing the first application? If so, explain whether they were compiler or run-time errors and what you did to fix them.

1.5.5 Making a JAR File

After we finish writing, compiling, and testing our application, we may want to distribute it. Java programs are typically distributed as **JARs**, which stands for Java ARchive. The JAR format allows the compression and aggregation of multiple files into one. The *.jar* extension is, by default, associated with the Java Virtual Machine. Thus, a user can double-click on a JAR file and launch its corresponding application directly. Most IDEs include a tool to easily create a JAR file. But you can also create a JAR file from the command line. The general syntax to create a JAR file is:

```
jar cf jarfile inputfile(s)
```

- The *c* and *f* characters are options. The *c* option means that we want to create a JAR file. The *f* option means that we want to send the output to a file.
- *jarfile* is the name of the JAR file that we want to create; it can be a different name from the name of our Java file(s).
- inputfile(s) is a space-separated list of files that we want to include in the JAR file. It typically includes the .class files and may include folders or files containing resources used in our application, such as images or sounds.

We can create a JAR file named *Greetingjar* from our *FirstProgram* application by typing the following at the command line:

```
jar cf Greeting.jar FirstProgram.class
```

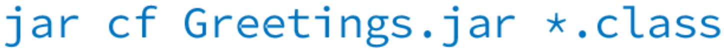
Note that the input file is the *. class* file, not the *.java* file. **Figure 1.5** shows the contents of the folder after executing the command above.

Figure 1.5 The Folder After the JAR File Is Created

Description

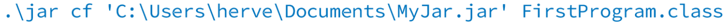
The file *Greetingjar* is created in the current folder, as shown in Figure 1.5.

If our application includes several Java files, we can use the wildcard character * to include all the corresponding *. class* files. The following creates a JAR file for all the *. class* files located in the current folder:



```
jar cf Greetings.jar *.class
```

If the *jar* command is not recognized, you can always go to the folder where it is (on the author’s laptop, it is *C:\Program Files\Java\jdk-17.0.1\bin)* and run the following command:

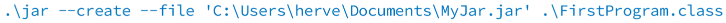


```
.\jar cf 'C:\Users\herve\Documents\MyJar.jar' FirstProgram.class
```

Description

As we can see above, we can also specify an absolute path for our destination *jar* file (here, we intend to place the JAR file in the *C:\Users\herve\Documents* folder).

Instead of *cf*, we can also use *create* and *file* as explicit options of the *jar* command as in:



```
.\jar --create --file 'C:\Users\herve\Documents\MyJar.jar' .\FirstProgram.class
```

Description

© Lois GoBe/Shutterstock

CHAPTER REVIEW

1.6 Chapter Summary

- Basic components of a computer include the CPU, memory, a hard disk, keyboard, and monitor.
- Each type of CPU has its own set of instructions for performing arithmetic and logical operations, moving data, and changing the order of execution of instructions.
- An operating system controls peripheral devices, supports multitasking, allocates memory to programs, and prevents the user from damaging the system.
- Computer networks link two or more computers so that they can share resources, such as files or printers.
- The Internet connects millions of computers around the world. Web servers deliver webpages to clients running Internet browsers.
- Binary numbers are composed of bits. Each bit has the value 0 or 1. A byte holds eight binary digits.
- To convert a binary number to a decimal number, multiply each digit in the binary number by $2^{\text{position}21}$, counting the rightmost position as position 1 and moving left through the number. Then add the products together.
- To convert a decimal number into a binary number, first find the largest power of 2 that is smaller than or equal to the decimal number, subtract that number from the decimal

number, then do the same with the remainder, and so on, until the decimal number reaches 0.

- Hexadecimal digits can be used to represent groups of four bits.
- The Unicode character set, which Java uses, can encode up to 65,536 characters using 16 bits per character.
- Machine language and assembly language are early forms of programming languages that require the programmer to write to the CPU's instruction set. Because this low-level programming is time consuming and difficult, and the programs are not portable to other CPU architectures, machine language and assembly language are rarely used.
- High-level languages are highly symbolic and somewhat portable. They can be compiled, interpreted, or as in the case of Java, converted to byte codes, which are interpreted at run time by the Java Virtual Machine.
- A good program is readable, easily maintainable, and reusable.
- Object-oriented programming uses classes to encapsulate data and the functions needed to manipulate that data. Objects are instantiated according to the class design. An advantage to object-oriented programming is the ability to reuse classes.
- Programs use a combination of sequential processing, method calls, selection, and iteration to control the order of execution of instructions. Performing operations in order, one after another, is called sequential processing. Temporarily executing other code, then returning, is called a method call. Selecting which code to execute based on the value of data is called selection. Repeating the same code on each item in a group of values is called iteration, or looping.
- Pseudocode allows a programmer to design a program without worrying about the syntax of the language.
- In programming, we often perform tasks for which there are standard methods of processing, called algorithms. For example, accumulating is a common programming operation that finds the sum of a group of numbers.

- Programming, in large part, is reducing a complex task to a set of subtasks that can be implemented by combining standard algorithms that use sequential processing, selection, and looping.
- Java source code is stored in a text file with an extension of *.java*. Compiling the code produces one or more *.class* files.
- An Integrated Development Environment (IDE) consists of a program editor, a compiler, and a run-time environment, integrated via a Graphical User Interface.
- Compiler errors are detected by the compiler and are usually caused by incorrect Java syntax or misspellings. Run-time errors are detected by the Java Virtual Machine and are usually caused by exceptions or incorrect use of classes. Logic errors occur during program execution and are caused by incorrect program design.

1.7 Exercises, Problems, and Projects

1.7.1 Multiple Choice Exercises

1. Which one of these is not an operating system?

    - Linux
    - Java
    - Windows
    - Android

6. Which one of these is not an application?

    - Word
    - Firefox
    - Linux
    - Excel

11. How many bits are in three bytes?

    - 3
    - 8
    - 24
    - 0

16. In a network, the computers providing services to the other computers are called

    - clients.
    - servers.
    - laptops.

20. A binary number ending with a 0

- is even.
- is odd.
- cannot tell.

24. A binary number ending with a 1

    - is even.
    - is odd.
    - cannot tell.

28. A binary number ending with two 0s

    - is a multiple of 4.
    - is not a multiple of 4.
    - cannot tell.

32. Using four bits, the largest positive binary number we can represent is 1111.

    - true
    - false

35. Which one of these is not a programming language?

    - C+ +
    - Java
    - Windows
    - Fortran

40. Which one of these is not an object-oriented programming language?

    - C
    - Java
    - C+ +
    - Smalltalk

45. What is the file extension for a Java source code file?

    - .java
    - .exe
    - .class

49. What is the file extension of a compiled Java program?

    - .java
    - .exe
    - .class

53. In order to compile a program named *Hello.java*, what do you type at the command line?

    - java Hello
    - java Hello.java
    - javac Hello
    - javac Hello.java

58. You have successfully compiled *Hello.java* into *Hello.class.* What do you type at the command line in order to run the application?

    - java Hello.class
    - java Hello
    - javac Hello
    - javac Hello.class

1.7.2 Converting Numbers

1. Convert the decimal number 67 into binary.
2. Convert the decimal number 1,564 into binary.
3. Convert the binary number 0001 0101 into decimal.
4. Convert the binary number 1101 0101 0101 into decimal.

5. Convert the binary number 0001 0101 into hexadecimal.
6. Convert the hexadecimal number D8F into binary.

1.7.3 General Questions

1. A RAM chip is organized as × 8 memory, i.e., each unit contains 8 bits, or a byte. There are 7 address pins on the chip. How many bytes does that memory chip contain?
2. If a CPU is rated at 3 GHz, how many clock cycles per second are performed?
3. If a CPU's clock cycles 2.6 billion times per second, what is the rating of the CPU in MHz?
4. Suppose we are using binary encoding to represent colors. For example, a black-and-white color system has only two colors and therefore needs only 1 bit to encode the color system as follows:

Bit	Color
0	black
I	white

   With 2 bits, we can encode four colors as follows:

Bit pattern	Color
00	black
01	red
10	blue
11	white

   With 5 bits, how many colors can we encode?
   With *n* bits (*n* being a positive integer), how many colors can we encode? (Express your answer as a function of *n*.)
5. In HTML, a color can be coded in the following hexadecimal notation: *#rrggbb,* where
   rr represents the amount of red in the color
   gg represents the amount of green in the color
   bb represents the amount of blue in the color

rr, gg, and *bb* vary between 00 and FF in hexadecimal notation, i.e., 0 and 255 in decimal equivalent notation. Give the decimal values of the red, green, and blue values in the color #33AB12.

6. RGB is a color system representing colors: R stands for red, G for green, and B for blue. A color can be coded as *rgb* where *r* is a number between 0 and 255 representing how much red there is in the color, *g* is a number between 0 and 255 representing how much green there is in the color, and *b* is a number between 0 and 255 representing how much blue there is in the color. The color gray is created by using the same value for *r*, *g*, and *b*. How many shades of gray are there?
7. List three benefits of the Java programming language.
8. What is the name of the Java compiler?
9. Write the pseudocode for a program that finds the product of two numbers.
10. Write the pseudocode for a program that finds the sums of the numbers input that are greater than or equal to 10 and the numbers input that are less than 10.

1.7.4 Technical Writing

1. List the benefits of having a Local Area Network versus standalone computer systems.
2. For one day, keep a diary of the computer applications that you use. Also note any features of the applications that you think should be improved or any features you'd like to see added.
3. You are looking at two computers with the following specifications, everything else being equal:

PC # 1	PC # 2
3.4-GHz CPU	3.3-GHz CPU

16 MB L2 cache	16 MB L2 cache
4 GB RAM	12 GB RAM
1 TB hard drive	1 TB hard drive
$399	$399

Which PC would you buy? Explain the reasoning behind your selection.

4. Go to Oracles Java site *(www.oracle.com/technetwork/java).* Explain what resources are available there for someone who wants to learn Java.

1.7.5 Group Project (for a group of 1, 2, or 3 students)

1. In the octal system (base 8), numbers are represented using digits from 0 to 7; a 0 is placed in front of the octal number to indicate that the octal system is being used. For instance, here are some examples of the equivalent of some octal numbers in the decimal system:

Octal	Decimal
000	0
001	1

007	7
010	8
011	9

In the hexadecimal system, numbers are represented using digits from 0 to 9 and letters A to F; 0x is placed in front of the hexadecimal number to indicate that the hexadecimal system is being used. For instance, here are some examples of the decimal equivalents of some hexadecimal numbers:

Hexadecimal	Decimal
0×0	0
0×1	1
0×9	9
0×A	10
0×B	11
0×F	15
0×10	16
0×11	17
0×1C	28

1. Convert 0xC3E (in hexadecimal notation) into an octal number.
2. Convert 0377 (in octal notation) into a hexadecimal number.
3. Discuss how, in general, you would convert a hexadecimal number into an octal number and an octal number into a hexadecimal number.

© Lois GoBe/Shutterstock.

CHAPTER 2

Programming Building Blocks—Java Basics

CHAPTER CONTENTS

Introduction
2.1 Java Application Structure
2.2 Data Types, Variables, and Constants

  2.2.1 Declaring Variables
  2.2.2 Integer Data Types
  2.2.3 Floating-Point Data Types
  2.2.4 Character Data Type
  2.2.5 Boolean Data Type
  2.2.6 The Assignment Operator, Initial Values, and Literals
  2.2.7 *String* Literals and Escape Sequences
  2.2.8 Text Blocks
  2.2.9 Constants

2.3 Expressions and Arithmetic Operators

    - 2.3.1 The Assignment Operator and Expressions
    - 2.3.2 Arithmetic Operators
    - 2.3.3 Operator Precedence
    - 2.3.4 **Programming Activity 1: Converting Inches to Centimeters**
    - 2.3.5 Integer Division and Modulus
    - 2.3.6 Division by Zero
    - 2.3.7 Mixed-Type Arithmetic and Type Casting
    - 2.3.8 Shortcut Operators

- 2.4 **Programming Activity 2: Temperature Conversion**
- 2.5 Chapter Summary
- 2.6 Exercises, Problems, and Projects

    - 2.6.1 Multiple Choice Exercises
    - 2.6.2 Reading and Understanding Code
    - 2.6.3 Fill In the Code
    - 2.6.4 Identifying Errors in Code
    - 2.6.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
    - 2.6.6 Write a Short Program
    - 2.6.7 Programming Projects
    - 2.6.8 Technical Writing

Introduction

If we boil it down to the basics, a program has two elements: instructions and data. The instructions tell the CPU what to do with the data. Typically, a program's structure will consist of the following operations:

1. Input the data.
2. Perform some processing on the data.
3. Output the results.

The data used by a program can come from a variety of sources. The user can enter data from the keyboard, for example, when we type a new document into a word processor. The program can read the data from a file, for example, when we load an existing document into the word processor. Or the program can generate the data randomly, for example, when a computer card game deals hands. Finally, some data is already known; for example, the number of hours in a day is 24, the number of days in December is 31, and the value of pi is 3.14159. This type of data is constant. The Java language provides a syntax for describing a program's data using keywords, symbolic names, and data types.

Although the data may be different in each execution of the program, the instructions stay the same. In a word processor, the words (data) are different from document to document, but the operation (instructions) of the word processor remains the same. When a line becomes full, for example, the word processor automatically wraps to the next line. It doesn't matter which words are on the line, only that the line is full. When we select a word and change the font to bold, it doesn't matter which word we select; it will become bold. Thus, a program's instructions (its algorithm) must be written to correctly handle any data it may receive.

We will write our programs by translating our algorithms into the basic operations that the computer can perform: input and output of data and various operations related to processing data, such as arithmetic calculations, comparisons of data and subsequent changes to the flow of control, and movement of data from one location in memory to another.

In this chapter, we’ll look at basic Java syntax for defining the data to be used in the program, performing calculations on that data, and outputting program results to the screen.

```
The area of the circle is 38.4844775
```

```
The area of the circle is 1256.636
```

2.1 Java Application Structure

Every Java program consists of at least one class. It is impossible to write a Java program that doesn’t use classes. Classes describe a logical entity that has data as well as methods (the instructions) to manipulate that data. An object is a physical instantiation of the class that contains specific data. In **Example 2.1** we provide a shell that contains the basic format of a Java application with a class name of *ShellApplication*. Our source code will use this format, changing the class name as appropriate.

Example 2.1 A Shell for a Java Application

```java
/* An application shell
   Anderson, Franceschi
*/
public class ShellApplication
{
   public static void main( String [ ] args ) // required
   {
      // write your code here
   }
}
```

Description

In Example 2.1, the numbers to the left of each line are not part of the program code; they are included here for our convenience. IDEs typically allow us to display line numbers.

From application to application, the name of the class, *ShellApplication*, will change, because we will want to name our class something meaningful that reflects its function. Each Java source code file must have the same name as the class name with a *.java* extension. In this case, the source file must be *ShellApplication.java*. Whatever name we select for a class must comply with the Java syntax for identifiers.

Java **identifiers** are symbolic names that we assign to classes, methods, and data. Identifiers must start with a **Java letter** and may contain any combination of letters and digits, but no spaces. A Java letter is any character in the range *a–z* or *A–Z*, the underscore (_), or the dollar sign ($), as well as many Unicode characters that are used as letters in other languages. Digits are any character between 0 and 9. The length of an identifier is essentially unlimited. Identifier names are case sensitive, so *Number1* and *number1* are considered to be different identifiers.

In addition, none of Java's **reserved words** can be used as identifiers. These reserved words, which are listed in Appendix A, consist of keywords used in Java instructions, as well as three special data values: *true*, *false*, and *null*. As of Java 9, the single underscore (_) is a reserved word and cannot be used as an identifier. Given that Java identifiers are case sensitive, note that it is legal to use *True* or *TRUE* as identifiers, but *true* is not a legal variable name. **Table 2.1** lists the rules for creating Java identifiers.

Rules for Creating Identifiers

Java Identifiers

- Must start with a Java letter (*A–Z*, *a–z*, _, $, or many Unicode characters)
- Can contain an almost unlimited number of letters and/or digits (0–9)
- Cannot contain spaces
- Are case sensitive
- Cannot be a Java reserved word

Description

The shell code in Example 2.1 uses four identifiers: *ShellApplication*, *main*, *String*, and *args*. The remainder of Example 2.1 consists of comments, Java keywords, and required punctuation.

The basic building block of a Java program is the **statement**.

A statement is terminated with a semicolon and can span several lines.

Any amount of **white space** is permitted between identifiers, Java keywords, operands, operators, and literals. White space characters are the space, tab, newline, carriage return, and a few other rarely used characters. Liberal use of white space makes our program more readable. It is good programming style to surround identifiers, operands, and operators with spaces and to skip lines between logical sections of the program.

A **block**, which consists of 0, 1, or more statements, starts with a left curly brace ({) and ends with a right curly brace (}). Blocks are required for class and method definitions and can be used anywhere else in the program that a statement is legal. Example 2.1 has two blocks: the class definition (lines 5 through 10) and the *main* method definition (lines 7 through 9). As we can see, nesting blocks within blocks is perfectly legal. The *main* block is nested completely within the class definition block.

SOFTWARE ENGINEERING TIP

Liberal use of white space makes your program more readable. It is good programming style to surround identifiers, operands, and operators with spaces and to skip lines between logical sections of the program.

SOFTWARE ENGINEERING TIP

Include a block comment at the beginning of each source file that identifies the author of the program and briefly describes the function of the program.

Comments document the operation of the program and are notes to ourselves and to other programmers who read our code. Comments are not compiled and can be coded in two ways. **Block comments** can span several lines; they begin with a forward slash-asterisk (/*) and end with an asterisk-forward slash (*/). Everything between the /* and the */ is ignored by the compiler. Note that there are no spaces between the asterisk and forward slash. Lines 1–3 in Example 2.1 are block comments and illustrate the good software engineering practice of providing at the beginning of our source code a few comments that identify ourselves as the author and briefly describe what the program does.

The second way to include comments in our code is to precede the comment with two forward slashes (//). There are no spaces between the forward slashes. The compiler ignores everything from the two forward slashes to the end of the line. In Example 2.1, the compiler ignores all of line 8, but only the part of line 6 after the two forward slashes.

Let's look at an example to get a sense of what a simple program looks like and to get a feel for how a program operates. **Example 2.2** calculates the area of a circle.

Example 2.2 Calculating the Area of a Circle

```
 1 /* Calculate the area of a circle
 2    Anderson, Franceschi
 3 */
 4
 5 public class AreaOfCircle
 6 {
 7   public static void main( String [ ] args )
 8   {
 9      // define the data we know
10      final double PI = 3.14159;
11
12      // define other data we will use
13      double radius = 3.5;
14
15      // perform the calculation and store the result
16      double area = PI * radius * radius;
17
18      // output the result
19      System.out.println( "The area of the circle is " + area );
20   }
21 }
```

Description

Figure 2.1a shows the output when the program is run with a radius of 3.5. To calculate the area of a circle with a different radius, replace the value 3.5 in line 13 with the new radius value. For example, to calculate the area of a circle with a radius of 20, change line 13 to

```
double radius = 20;
```

Then recompile the program and run it again. **Figure 2.1b** shows the output for a radius of 20.

Figure 2.1a Output from Example 2.2 with a Radius of 3.5

Description

Figure 2.1b Output from Example 2.2 with a Radius of 20

Description

We can see that Example 2.2 has the basic elements that we saw in the *ShellApplication* (Example 2.1). We have added some statements in lines 9 through 19 that do the work of the program. First, we identify the data we will need. To calculate the area of a circle, we use the formula (πr^2). We know the value of π (3.14159), so we store that value in a memory location we name PI (line 10). We also need places in memory to hold the radius and the area. In line 13 we name the radius and, we give the radius a value; here we have chosen 3.5, but our program will need to work with any value for the radius.

Now we're ready to calculate the area. We want this program to output correct results with any radius, so we need to write the algorithm of the program using the formula for calculating a circle's area given above. Java provides arithmetic operators for performing calculations. We use Java's multiplication operator (*) in line 16 to multiply PI times the radius times the radius and store the result into the memory location we named *area*. Now we're ready to output the result. On line 19, we write a message that includes the *area* value we calculated.

With Java 9, we have another option to running Java code. If we just want to try out some Java code without writing a complete application, we can use the **jshell** utility. The jshell utility is an REPL (Read, Evaluate, Print, Loop) tool that allows us to type small amounts of code, called snippets, and see the immediate result without compiling. Some IDEs (Integrated Development Environments) already provide an interactive utility like jshell. For example, jGRASP (*jgrasp.org*) provides an Interactions tool. We will use jshell to illustrate many concepts in this chapter. You will find jshell in the *bin* folder where the Java JDK is stored. If you have already updated your classpath environment variable for javac and java, then it is automatically set for jshell because these three programs are in the same folder. Consult your operating system instructions for details about setting the classpath.

To use jshell, open a command line application and launch jshell. To try out some code, simply type it at the prompt. To see a complete list of jshell commands, type */help*. As you will see, jshell commands start with a forward slash (/). To exit jshell, type */exit*.

```
Please choose a language:
    1 - Java
    2 - Python
    3 - C++
```

```
The value of constant ZORRO is Z
The value of constant PI is 3.14159
The number of days in a leap year is 366
The number of days in a non-leap year is 365
```

2.2 Data Types, Variables, and Constants

In Example 2.2, we used as data the value of PI and the radius, and we calculated the area of the circle. For each of these values, we assigned a name. We also used the Java keyword *double,* which defines the **data type** of the data. The keyword *double* means that the value will be a floating-point number.

Java allows us to refer to the data in a program by defining **variables**, which are named locations in memory where we can store values. A variable can store one data value at a time, but that value might change as the program executes, and it might change from one execution of the program to the next. The real advantage of using variables is that we can name a variable, assign it a value, and subsequently refer to the name of the variable in an expression rather than hard coding the specific value.

When we use a named variable, we need to tell the compiler which kind of data we will store in the variable. We do this by giving a data type for each variable.

Java supports eight primitive data types: *byte, short, int, long, float, double, char,* and *boolean*. They are called primitive data types because they are part of the core Java language.

The data type we specify for a variable tells the compiler how much memory to allocate and the format in which to store the data. For example, if we specify that a data item is an *int*, then the compiler will allocate four bytes of memory and store its value as a 32-bit signed binary number. If, however, we specify that a data item is a *double* (a double-precision floating-point number), then the compiler will allocate 8 bytes of memory and store its value as a double-precision IEEE 754 floating-point number.

SOFTWARE ENGINEERING TIP

When selecting identifiers, choose meaningful names that reflect the use of the identifier in the program; this will make your code self-documented. Use as many characters as necessary to make the identifier clear, but avoid extremely long identifiers. Also, for clarity in your program logic, avoid identifiers that resemble Java keywords.

Once we declare a data type for a data item, the compiler will monitor our use of that data item. If we attempt to perform operations that are not allowed for that type or are not compatible with that type, the compiler will generate an error. Because the Java compiler monitors the operations on each data item, Java is called a **strongly typed language**.

Take care in selecting identifiers for your programs. The identifiers should be meaningful and should reflect the data that will be stored in a variable, the concept encapsulated by a class, or the function of a method. For example, the identifier *age* clearly indicates that the variable will hold the age of a person. When we select meaningful variable names, the logic of our program is more easily understood, and we are less likely to introduce errors. Sometimes, it may be necessary to create a long identifier in order to clearly indicate its use, for example, *numberOfStudentsWhoPassedCS1*. Although the length of identifiers is essentially unlimited, avoid creating extremely long identifiers because they are more cumbersome to use. Also, the longer the identifier, the more likely we are to make typos when entering the identifier into our program. Finally, although it is legal to use identifiers, such as *TRUE*, which differ from Java keywords only in case, it isn't a good idea because they easily can be confused with Java keywords, making the program logic less clear.

2.2.1 Declaring Variables

Every variable must be given a name and a data type before it can be used. This is called **declaring a variable**.

The syntax for declaring a variable is:

```
dataType identifier; // this declares one variable
```

Description

or

```
dataType identifier1, identifier2, ...; // this declares multiple
                                        // variables of the same
                                        // data type
```

Description

Note that a comma follows each identifier in the list except the last identifier, which is followed by a semicolon.

By convention, the identifiers for variable names should start with a lowercase letter. If the variable name consists of more than one word, then each word after the first should begin with a capital letter. For example, these identifiers are conventional Java variable names: *number1*, *highScore*, *booksToRead*, *ageInYears*, and *xAxis*. Underscores conventionally are not used in variable names; they are reserved for the identifiers of constants, as we shall discuss later in the chapter. Similarly, do not use dollar signs to begin variable names. The dollar sign is reserved for the first letter of programmatically generated variable names—that is, variable names generated by software, not people. Although this may sound arbitrary now, the value of following these conventions will become clearer as we gain more experience in Java and our programs become more complex.

SOFTWARE ENGINEERING TIP

Begin variable names with a lowercase letter. If the variable name consists of more than one word, begin each word after the first with a capital letter. Avoid underscores in variable names, and do not begin a variable name with a dollar sign.

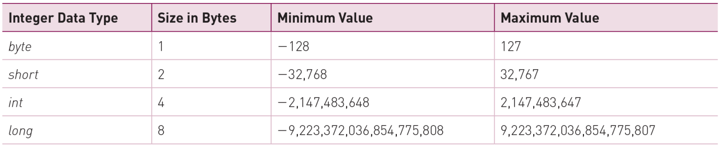
2.2.2 Integer Data Types

An integer data type is one that evaluates to a positive or negative whole number. Java provides four integer data types: *int*, *short*, *long*, and *byte*.

The *int*, *short*, *long*, and *byte* types differ in the number of bytes of memory allocated to store each type and, therefore, the maximum and minimum values that can be stored in a variable of that type. All of Java's integer types are signed, meaning that they can be positive or negative; the high-order, or leftmost, bit is reserved for the sign.

Table 2.2 summarizes the integer data types, their sizes in memory, and their maximum and minimum values.

Integer Data Types



Integer Data Type	Size in Bytes	Minimum Value	Maximum Value
byte	1	−128	127
short	2	−32,768	32,767
int	4	−2,147,483,648	2,147,483,647
long	8	−9,223,372,036,854,775,808	9,223,372,036,854,775,807

Description

In most applications, the *int* type will be sufficient for our needs, since it can store positive and negative numbers into the 2 billion range. The *short* and *byte* data types typically are used only when memory space is critical, and the *long* data type is needed only for data values larger than 2 billion.

Let's look at some examples of integer variable declarations.

Note that the variables' names clearly indicate the data that the variables will hold.

```
int testGrade;
int numPlayers, highScore, diceRoll;
short xCoordinate, yCoordinate;
long cityPopulation;
byte ageInYears;
```

Description

2.2.3 Floating-Point Data Types

Floating-point data types store numbers with fractional parts. Java supports two floating-point data types: the single-precision *float* and the double-precision *double*.

REFERENCE POINT

Floating-point numbers are stored using the IEEE 754 standard, which is discussed in Appendix E.

The two types differ in the amount of memory allocated and the size and precision of the number that can be represented. The single-precision type (*float*) is stored in 32 bits, while the double-precision type (*double*) is stored in 64 bits. *Floats* and *doubles* can be positive or negative.

Table 2.3 summarizes Java's floating-point data types, their sizes in memory, and their maximum and minimum positive nonzero values.

Floating-Point Data Types

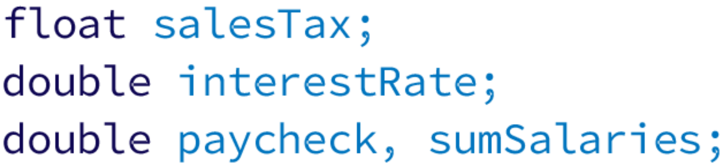
Floating-Point Data Type	Size in Bytes	Minimum Positive Nonzero Value	Maximum Value
float	4	1.4E-45	3.4028235E38
double	8	4.9E-324	1.7976931348623157E308

Description

Because of its greater precision, the *double* data type is usually preferred over the *float* data type. However, for calculations not requiring such precision, *floats* are often used because they require less memory.

Although integers can be stored as *doubles* or *floats*, it isn't advisable to do so because floating-point numbers require more processing time for calculations.

Let's look at a few examples of floating-point variable declarations:



```
float salesTax;
double interestRate;
double paycheck, sumSalaries;
```

Description

2.2.4 Character Data Type

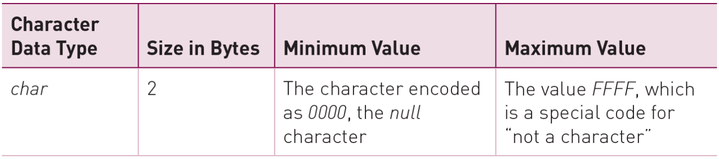
The *char* data type stores one Unicode character. Because Unicode characters are encoded as unsigned numbers using 16 bits, a *char* variable is stored in two bytes of memory.

REFERENCE POINT

The encoding of ASCII and Unicode characters is discussed in Appendix C.

Table 2.4 shows the size of the *char* data type, as well as the minimum and maximum values. The maximum value is the unsigned hexadecimal number *FFFF*, which is reserved as a special code for "not a character."

The Character Data Type

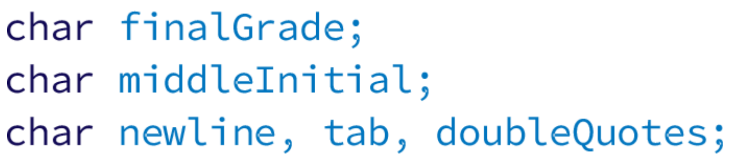


Character Data Type	Size in Bytes	Minimum Value	Maximum Value
char	2	The character encoded as *0000*, the *null* character	The value *FFFF*, which is a special code for "not a character"

Description

Obviously, since the *char* data type can store only a single character, such as a *K***,** a *char* variable is not useful for storing names, titles, or other text data. For text data, Java provides a *String* class, which we'll discuss later in this chapter.

Here are a few declarations of *char* variables:



```
char finalGrade;
char middleInitial;
char newline, tab, doubleQuotes;
```

Description

2.2.5 Boolean Data Type

The *boolean* data type can store only two values, which are expressed using the Java reserved words *true* and *false*, as shown in **Table 2.5**.

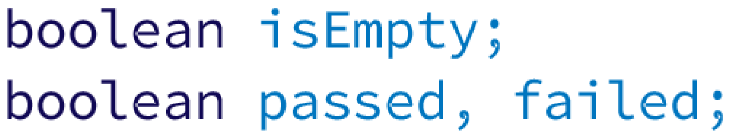
The *boolean* Data Type

***boolean* Data Type**	**Possible Values**
boolean	*true* *false*

Description

Booleans are typically used for decision making and for controlling the order of execution of a program.

Here are examples of declarations of *boolean* variables:



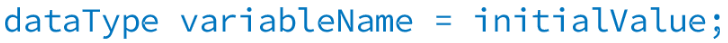
```
boolean isEmpty;
boolean passed, failed;
```

Description

2.2.6 The Assignment Operator, Initial Values, and

Literals

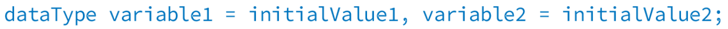
When we declare a variable, we can also assign an initial value for the data. To do that, we use the **assignment operator** (5) with the following syntax:



```
dataType variableName = initialValue;
```

Description

This statement is read as "*variableName* **gets** *initialValue*".
or

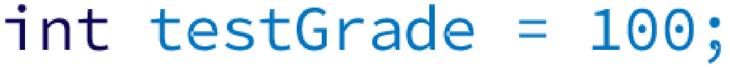


```
dataType variable1 = initialValue1, variable2 = initialValue2;
```

Description

Notice that assignment is right to left. The initial value is assigned to the variable.

One way to specify the initial value is by using a **literal value.** In the following statement, the value *100* is an *int* literal value, which is assigned to the variable *testGrade*.



```
int testGrade = 100;
```

Description

COMMON ERROR TRAP

Although Unicode characters occupy two bytes in memory, they still represent a single character. Therefore, a literal value must also represent only one character.

Table 2.6 summarizes the legal characters in literals for all primitive data types.

Literal Formats for Java Data Types

Data Type	**Literal Format**
int, short, byte	Optional initial sign (+ or –)

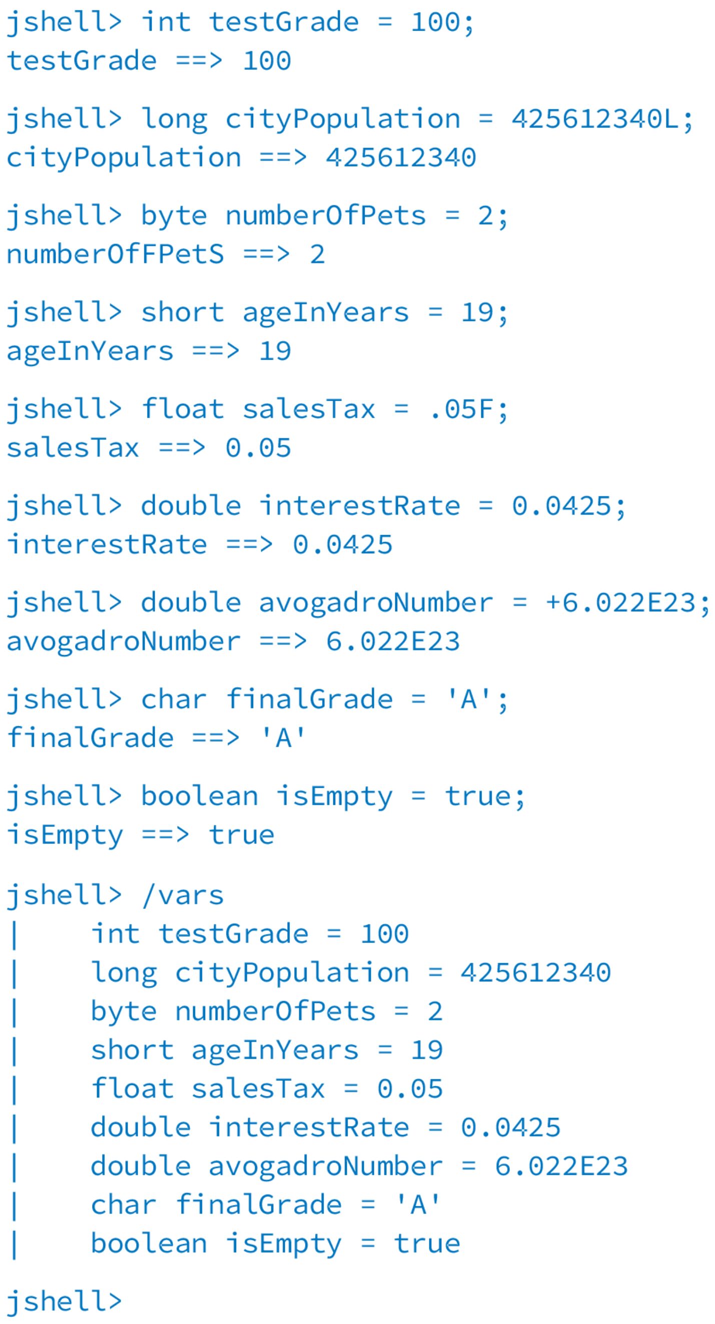
	followed by digits 0–9 in any combination. A literal in this format is an *int* literal; however, an *int* literal may be assigned to a *byte* or *short* variable if the literal is a legal value for the assigned data type. An integer literal that begins with a 0 digit is considered to be an octal number (base 8) and the remaining digits must be 0–7. An integer literal that begins with 0x is considered to be a hexadecimal number (base 16) and the remaining digits must be 0–9 or A–F.
long	Optional initial sign (+ or –) followed by digits 0–9 in any combination, terminated with an *L* or *l*. It's preferable to use the capital *L*, because the lowercase *l* can be confused with the number *1*. An integer literal that begins with a 0 digit is considered to be an octal number (base 8) and the remaining digits must be 0–7. An integer literal that begins with 0x is considered to be a hexadecimal number (base 16) and the remaining digits must be 0–9 or A–F.
float	Optional initial sign (+ or –) followed by a floating-point number in fixed or scientific format, terminated by an *F* or *f*.
double	Optional initial sign (+ or –) followed by a floating-point number in fixed or scientific format.
char	■ Any printable character enclosed in single quotes. ■ A decimal value from 0 to 65,535. ■ '\unnnn' where nnnn are hexadecimal digits. ■ '\m', where \m is an escape sequence. For example, '\n' represents a newline, and '\t' represents a tab character.
boolean	*true* or *false*

Description

Notice in Table 2.6 under the literal format for *char*, that *\n* and *\t* can be used to format output. We'll discuss these and other escape sequences in the next section of this chapter.

Example 2.3 shows a jshell session where we declare variables and assign a value using a literal. We then execute the jshell */vars* command to see our variables and their values.

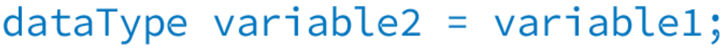
Example 2.3 Declaring and Initializing Variables



```
jshell> int testGrade = 100;
testGrade ==> 100

jshell> long cityPopulation = 425612340L;
cityPopulation ==> 425612340

jshell> byte numberOfPets = 2;
numberOfFPetS ==> 2

jshell> short ageInYears = 19;
ageInYears ==> 19

jshell> float salesTax = .05F;
salesTax ==> 0.05

jshell> double interestRate = 0.0425;
interestRate ==> 0.0425

jshell> double avogadroNumber = +6.022E23;
avogadroNumber ==> 6.022E23

jshell> char finalGrade = 'A';
finalGrade ==> 'A'

jshell> boolean isEmpty = true;
isEmpty ==> true

jshell> /vars
|    int testGrade = 100
|    long cityPopulation = 425612340
|    byte numberOfPets = 2
|    short ageInYears = 19
|    float salesTax = 0.05
|    double interestRate = 0.0425
|    double avogadroNumber = 6.022E23
|    char finalGrade = 'A'
|    boolean isEmpty = true

jshell>
```

Description

Another way to specify an initial value for a variable is to assign the variable the value of another variable, using this syntax:



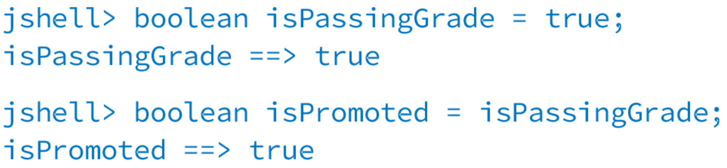
```
dataType variable2 = variable1;
```

Description

Two things need to be true for this assignment to work:

- *variable1* needs to be declared and assigned a value before this statement appears in the source code.
- *variable1* and *variable2* need to be compatible data types; in other words, the precision of *variable1* must be lower than or equal to that of *variable2*.

For example, in these statements:

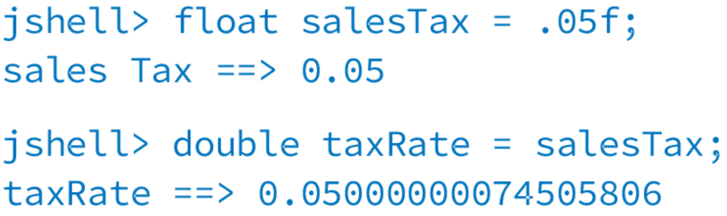


```
jshell> boolean isPassingGrade = true;
isPassingGrade ==> true

jshell> boolean isPromoted = isPassingGrade;
isPromoted ==> true
```

Description

isPassingGrade is given an initial value of *true*. Then *isPromoted* is assigned the value already given to *isPassingGrade*. Thus, *isPromoted* is also assigned the initial value *true*. If *isPassingGrade* were assigned the initial value *false*, then *isPromoted* would also be assigned the initial value *false*.

And in these statements,



```
jshell> float salesTax = .05f;
sales Tax ==> 0.05

jshell> double taxRate = salesTax;
taxRate ==> 0.05000000074505806
```

Description

the initial value of .05 is assigned to *salesTax* and then to *taxRate*. It’s legal to assign a *float* value to a *double*, because all values that can be stored as *floats* are also valid *double* values. Note that after assigning the *float* value to the *double,* that *taxRate* has some extraneous digits. This is due to rounding errors when converting the *float* value to a *double*. Essentially, the two values are equivalent.

However, assigning a *double* value to a *float* variable is *not* valid:

```
jshell> double taxRate = .05;
taxRate ==> 0.05

jshell> float salesTax = taxRate;
|  Error:
|  incompatible types: possible lossy conversion from double to float
|  float salesTax = taxRate;
|                   ^-----^
```

Description

Even though .05 is a valid *float* value, the compiler will generate a “`possible lossy conversion`” error similar to the error shown in jshell above.

Similarly, we can assign a lower-precision integer value to a higher-precision integer variable.

Table 2.7 summarizes compatible data types; a variable or literal of any type in the right column can be assigned to a variable of the data type in the left column.

Valid Data Types for Assignment

Data Type	**Compatible Data Types**
byte	*byte*
short	*byte, short*
int	*byte, short, int, char*
long	*byte, short, int, char, long*
float	*byte, short, int, char, long, float*
double	*byte, short, int, char, long, float, double*
boolean	*boolean*
char	*char*

Description

Variables need to be declared before they can be used in our program, but be careful to declare each variable only once; that is, specify the data type of the variable only the first time that variable is used in the program. If we attempt to declare a variable that has already been declared, as in the following statements:

```
double twoCents;
double twoCents = 2; // incorrect, second declaration of twoCents
```

Description

we will receive a compiler error similar to the following:

```
variable twoCents is already defined in method
```

COMMON ERROR TRAP

Declare each variable only once, the first time the variable is used. After the variable has been declared, its data type cannot be changed.

Similarly, once we have declared a variable, we cannot change its data type. Thus, these statements:

```
double cashInHand;
int cashInHand; // incorrect, data type cannot be changed
```

Description

will generate a compiler error similar to the following:

```
variable cashInHand is already defined in method
```

CODE IN ACTION

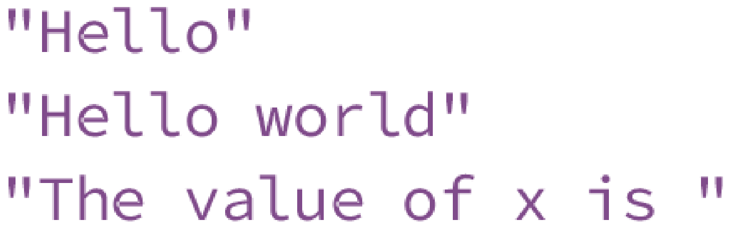
Within the online resources, you will find a movie showing a step-by-step illustration of declaring variables and assigning initial values. Click on this chapter's link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

2.2.7 *String* Literals and Escape Sequences

In addition to literals for all the primitive data types, Java also supports *String* literals. *String* literals are objects of Java's *String* class.

A ***String* literal** is a sequence of characters enclosed by double quotes. One set of quotes "opens" the *String* literal and the second set of quotes "closes" the literal. For example, these are all *String* literals:



```
"Hello"
"Hello world"
"The value of x is "
```

Description

We used *String* literals in output statements in Example 2.2 to label the data that we printed:

```
System.out.println( "The area of the circle is " + area );
```

The `+` operator is the ***String* concatenation operator**. Among other uses, the concatenation operator allows us to print the values of variables along with *String* literals. As we can see from the output of Example 2.2, the characters in the *String* literal are output exactly as typed, whereas the variable *area* is replaced by its current value.

SOFTWARE ENGINEERING TIP

Add a space to the end of a *String* literal before concatenating a value for more readable output.

String literals using double quotes cannot extend over more than one line. If the compiler finds a newline character in the middle of a *String* literal, it will generate a compiler error. For example, the following statement is not valid:

```
System.out.println( "Never pass a water fountain
                     without taking a drink." );
```

Description

COMMON ERROR TRAP

All open quotes for a *String* literal should be matched with a set of closing quotes, and the closing quotes must appear before the line ends.

In fact, that statement will generate errors:

```
jshell> System.out.println( "Never pass a water fountain
|  Error:
|  unclosed string literal
|  System.out.println( "Never pass a water fountain
|                      ^
```

Description

If we have a long *String* to print, break it into several strings and use the concatenation operator. This statement is a correction of the previous invalid statement:

```
jshell> System.out.println( "Never pass a water fountain"
 ... >                       + "without taking a drink." );
Never pass a water fountain without taking a drink.
```

Description

Another common programming error is omitting the closing quotes. Be sure that all open quotes have matching closing quotes on the same line.

Now that we know that quotes open and close *String* literals, how can we define a literal that includes quotes? This statement, which has quotes inside a *String* literal

```
System.out.println( "She said, "Java is fun".");  // illegal quotes
                                                  // within literal
```

Description

generates this error:

```
jshell> System.out.println( "She said, "Java is fun"." );
|  Error:
|  ')' expected
|  System.out.println( "She said, "Java is fun"." );
|                                  ^
```

Description

Java provides a set of escape sequences that can be used to include a special character within *String* and *char* literals. The escape sequences *\n, \t, \b, \r,* and *\f* are nonprintable characters. **Table 2.8** lists the Java escape sequences.

Java Escape Sequences

Character	**Escape Sequence**
newline	\n
tab	\t
double quotes	\"
single quote	\'
backslash	\\
backspace	\b
carriage return	\r
form feed	\f

Description

Using the \" escape character, we can now output embedded quotes as:

```
jshell> System.out.println( "She said, \"Java is fun\"." );
She said, "Java is fun".
```

Description

We can force part of the output to the next line by inserting a *\n* escape character, as in:

```
jshell> System.out.println( "One potato\nTwo potatoes" );
One potato
Two potatoes
```

Description

We will find that the *\n* escape character is useful for skipping lines in output.

Finally, we can format output by inserting *\t* where we want a tab, as in:

```
jshell> System.out.println( "\tTabs can make the output easier to read." );
        Tabs can make the output easier to read.
```

Description

2.2.8 Text Blocks

Text blocks, *String* literals delimited by **triple quotes**, allow us to define literal *Strings* over several lines without using the concatenation operator. In addition to being convenient, using text blocks can improve the readability of our code. At lines 9 to 14 of **Example 2.4**, we define the *String menu* using a text block. Furthermore, we can use escape sequences, such as \t, \n, \\, \”, and so on, within a text block. **Figure 2.2** shows the output of Example 2.4.

Example 2.4 Using Text Blocks

```
/* Using Text Blocks
   Anderson, Franceschi
*/

public class TextBlock
{
  public static void main( String [ ] args )
  {
    String menu = """
      Please choose a language:
      \t1 - Java
      \t2 - Python
      \t3 - C++
      """;

    System.out.println( menu );
  }
}
```

Description

Figure 2.2 Output of Example 2.4

Description

2.2.9 Constants

Sometimes we know the value of a data item, and we know that its value will not (and should not) change during program execution, nor is it likely to change from one execution of the program to another. In this case, it is a good software engineering practice to define that data item as a **constant**.

Defining constants uses the same syntax as declaring variables, except that the data type is preceded by the keyword *final*.

```
final dataType CONSTANT_IDENTIFIER = assignedValue;
```

Assigning a value is optional when the constant is defined, but we must assign a value before the constant is used in the program. Also, once the constant has been assigned a value, its value cannot be changed (reassigned) later in the program. Any attempt by our program to change the value of a constant will generate the following compiler error:

```
cannot assign a value to final variable
```

Think of this as a service of the compiler in preventing our program from unintentionally corrupting its data.

By convention, *CONSTANT_IDENTIFIER* consists of all capital letters, and embedded words are separated by an underscore. This makes constants stand out in the code and easily identified as constants. Also, constants are usually defined at the top of a program where their values can be seen easily.

Example 2.5 shows how to use constants in a program.

Example 2.5 Using Constants

```
 1 /* Constants Class
 2    Anderson, Franceschi
 3 */
 4
 5 public class Constants
 6 {
 7     public static void main( String [ ] args )
 8     {
 9          final char ZORRO = 'Z';
10          final double PI = 3.14159;
11          final int DAYS_IN_LEAP_YEAR = 366, DAYS_IN_NON_LEAP_YEAR = 365;
12
13          System.out.println( "The value of constant ZORRO is " + ZORRO );
14          System.out.println( "The value of constant PI is " + PI );
15          System.out.println( "The number of days in a leap year is "
16                                     + DAYS_IN_LEAP_YEAR );
17          System.out.println( "The number of days in a non-leap year is "
18                                     + DAYS_IN_NON_LEAP_YEAR );
19
20          // PI = 3.14;
21          // The statement above would generate a compiler error
22          // You cannot change the value of a constant
23     }
24 }
```

Description

Lines 9, 10, and 11 define four constants. On line 11, note that both *DAYS_IN_LEAP_YEAR* and *DAYS_IN_NON_LEAP_YEAR* are constants. We don’t need to repeat the keyword final to define two (or more) constants of the same data types. Lines 13 to 18 output the values of the four constants. If line 20 were not commented out, it would generate a compiler error because once a constant is assigned a value, its value cannot be changed. **Figure 2.3** shows the output of Example 2.5.

Figure 2.3 Output of Example 2.5

Description

SOFTWARE ENGINEERING TIP

Use all capital letters for a constant’s identifier; separate words with an underscore (_). Declare constants at the top of the program so their value can be seen easily.

SOFTWARE ENGINEERING TIP

Declare as a constant any data that should not change during program execution. The compiler will then flag any attempts by your program to change the value of the constant, thus preventing any unintentional corruption of the data.

Constants can make our code more readable: PI is more meaningful than 3.14159 when used inside an arithmetic expression. Another advantage of using constants is to keep programmers from making logic errors: Let’s say we set a constant to a particular value and it is used at various places throughout the code (for instance, a constant representing a tax rate); we then discover that the value of that constant needs to be changed. All we have to do is make the change in one place, most likely at the beginning of the code. If we had to change the value at many places throughout the code, that could very well result in logic errors or typos.

with these end-of-chapter questions

2.6.1	Multiple Choice Questions 1, 2
2.6.2	Reading and Understanding Code Questions 4, 5, 6
2.6.3	Fill In the Code Questions 23, 24, 25, 26
2.6.4	Identifying Errors in Code Questions 33, 34, 38, 39

2.6.5	Debugging Area Questions 40, 41
2.6.6	Write a Short Program Question 46
2.6.8	Technical Writing Question 52

```
The cost of lunch is $6.84
Your age in 2025 is 20
The calories in 3 apples is 381
The mileage is 28.078947368421055 miles per gallon.
```

```
5.2 inches are equivalent to 13.208 centimeters.
```

```
There are 4 quarters in 113 pennies
There are 13 pennies left over
The monthly salary is 4166.666666666667
```

```
The value of result1 is Infinity
The value of result2 is NaN
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivisionByZero.main(DivisionByZero.java:15)
```

```
At the beginning, a is 6
Increment a with prefix notation: 7
In the end, a is 7

At the beginning, b is 2
Increment b with postfix notation: 2
In the end, b is 3
```

```
a is 5
After a += 10; a is 15
After a -= 3; a is 12
After a *= 2; a is 24
After a /= 6; a is 4
After a %= 3; a is 1
```

2.3 Expressions and Arithmetic Operators

2.3.1 The Assignment Operator and Expressions

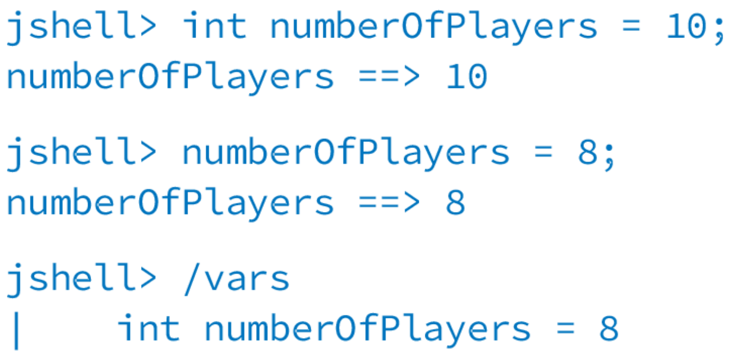
In a previous section, we mentioned using the assignment operator to assign initial values to variables and constants. Now let’s look at the assignment operator in more detail.
The syntax for the assignment operator is:

```
target = expression;
```

Description

An expression consists of operators and operands that evaluate to a single value. The value of the expression is then assigned to *target* (*target* gets *expression*), which must be a variable or a constant having a data type compatible with the value of the expression.

If *target* is a variable, the value of the expression replaces any previous value the variable was holding. For example, let's look at these instructions:

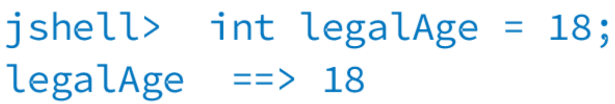


```
jshell> int numberOfPlayers = 10;
numberOfPlayers ==> 10

jshell> numberOfPlayers = 8;
numberOfPlayers ==> 8

jshell> /vars
|    int numberOfPlayers = 8
```

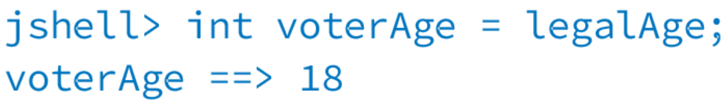
Description

The first instruction declares an *int* named *numberOfPlayers*. This allocates four bytes in memory to a variable named *numberOfPlayers* and stores the value 10 in that variable. Then, the second statement changes the value stored in the variable *numberOfPlayers* to 8. The previous value, 10, is discarded. Note that we do not repeat "*int*." As mentioned earlier, we declare the data type only the first time we use a variable.

An expression can be a single variable name or a literal of any type, in which case, the value of the expression is simply the value of the variable or the literal. For example, in these statements,



```
jshell>  int legalAge = 18;
legalAge  ==> 18
```

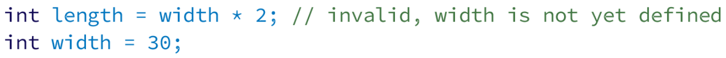


```
jshell> int voterAge = legalAge;
voterAge ==> 18
```

Description

the literal 18 is an expression. Its value is 18, which is assigned to the variable *legalAge*. Then, in the second statement, *legalAge* is an expression, whose value is 18. Thus, the value 18 is assigned to *voterAge*. So after these statements have been executed, both *legalAge* and *voterAge* will have the value 18.

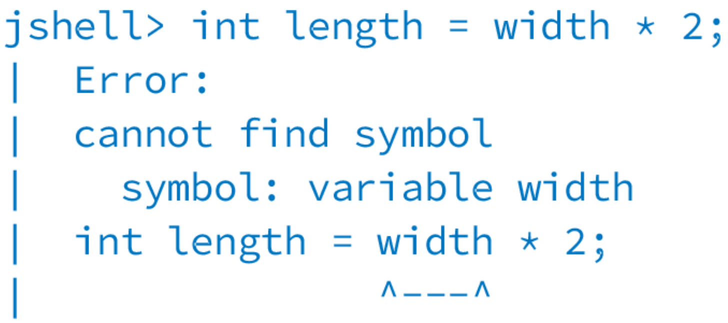
One restriction, however, is that an assignment expression cannot include another variable unless that variable has been defined previously. The statement defining the *length* variable that follows is **invalid**, because it refers to *width*, which is not defined until the next line.



```
int length = width * 2; // invalid, width is not yet defined
int width = 30;
```

Description

The compiler will flag the statement defining *length* as an error with a message similar to the one in the jshell session below



```
jshell> int length = width * 2;
|   Error:
|   cannot find symbol
|     symbol: variable width
|   int length = width * 2;
|                ^---^
```

Description

because *width* has not yet been defined.

An expression can be quite complex, consisting of multiple variables, constants, literals, and operators. Before we can look at examples of more complex expressions, however, we need to discuss the **arithmetic operators**.

2.3.2 Arithmetic Operators

Java’s arithmetic operators are used for performing calculations on numeric data. Some of these operators are shown in **Table 2.9**.

Arithmetic Operators

Operator	Operation
1	addition
2	subtraction
*	multiplication
/	division
%	modulus (remainder after division)

Description

All these operators take two operands; thus, they are called **binary operators**. Each operand is an expression.

In **Example 2.6**, we make a variety of calculations to demonstrate the addition, subtraction, multiplication, and division arithmetic operators. We will discuss integer division and the modulus operator later in the chapter. The output from this program is shown in **Figure 2.4**.

Example 2.6 Using Arithmetic Operators

```
 1 /* Arithmetic Operators
 2    Anderson, Franceschi
 3 */
 4
 5 public class ArithmeticOperators
 6 {
 7   public static void main( String [ ] args )
 8   {
 9      // calculate the cost of lunch
10      double salad = 5.95;
11      double water = .89;
12      System.out.println( "The cost of lunch is $"
13                          + ( salad + water ) );
14
15      // calculate your age as of a certain year
16      int targetYear = 2025;
17      int birthYear = 2005;
18      System.out.println( "Your age in " + targetYear + " is "
19                          + ( targetYear - birthYear ) );
20
21      // calculate the total calories of apples
22      int caloriesPerApple = 127;
23      int numberOfApples = 3;
24      System.out.println( "The calories in " + numberOfApples
25                          + " apples is " +
26                          + ( caloriesPerApple * numberOfApples ) );
27
28      // calculate miles per gallon
29      double miles = 426.8;
30      double gallons = 15.2;
31      double mileage = miles / gallons;
32      System.out.println( "The mileage is "
33                          + mileage + " miles per gallon." );
34   }
35 }
```

Description

Figure 2.4 Output from Example 2.6

Description

SOFTWARE ENGINEERING TIP

For readable code, insert a space between operators and operands.

Example 2.6 demonstrates a number of small operations. To calculate a total price (lines 12 and 13), we add the individual prices. To calculate an age (lines 18 and 19), we subtract the birth year from the target year. To calculate the number of calories in multiple apples (lines 24–26), we multiply the number of calories in one apple by the number of apples. We calculate miles per gallon by dividing the number of miles driven by the number of gallons of gas used (line 31). Note that we can either store the result in another variable, as we did in line 31, and subsequently output the result (lines 32–33), or we can output the result of the calculation directly by writing the expression in the *System.out.println* statement, as we did in the other calculations in this example.

SOFTWARE ENGINEERING TIP

Developing and testing your code in steps makes it easier to find and fix errors.

2.3.3 Operator Precedence

The statements in Example 2.6 perform simple calculations, but what if we want to make more complex calculations using several operations, such as calculating how much money we have in coins? Let’s say we have two quarters, three dimes, and two nickels. To calculate the value of these coins in pennies, we might use this expression:

```
int pennies = 2 * 25 + 3 * 10 + 2 * 5;
```

In which order should the computer do the calculation? If the value of the expression were calculated left to right, then the result would be

```
= 2 * 25 + 3 * 10 + 2 * 5
=   50   + 3 * 10 + 2 * 5
=        53   * 10 + 2 * 5
=            530    + 2 * 5
=                  532 * 5
=                     2660
```

Description

Clearly, 2,660 pennies is not the right answer. To calculate the correct number of pennies, the multiplications should be performed first, then the additions. This, in fact, is the order in which Java will calculate the preceding expression.

The Java compiler follows a set of rules called **operator precedence** to determine the order in which the operations should be performed.

Table 2.10 provides the order of precedence of the operators we’ve discussed so far. The operators in the first row—parentheses—are evaluated first, then the operators in the second row (*, /, %) are evaluated, and so on with the operators in each row. When two or more operators on the same level appear in the same expression, the order of evaluation is left to right, except for the assignment operator, which is evaluated right to left.

Operator Precedence

Operator Hierarchy	Order of Same-Statement Evaluation	Operation
()	left to right	parentheses for explicit grouping
*, /, %	left to right	multiplication, division, modulus
+, −	left to right	addition, subtraction
5	right to left	assignment

Description

As we introduce more operators, we’ll add them to the Order of Precedence chart. The complete chart is provided in Appendix B.

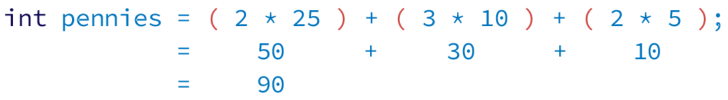
Using Table 2.10 as a guide, let’s recalculate the number of pennies:

```
int pennies = 2 * 25 +  3 * 10 + 2 * 5;
            =   50    +   30    +   10
            = 90
```

Description

As we can see, *90* is the correct number of pennies in two quarters, three dimes, and two nickels.

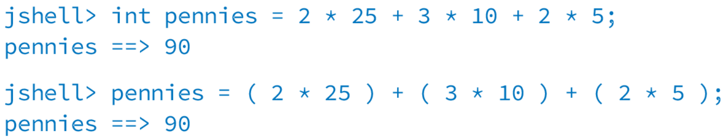
We also could have used parentheses to clearly display the order of calculation. For example,



```
int pennies = ( 2 * 25 ) + ( 3 * 10 ) + ( 2 * 5 );
            =     50     +     30     +     10
            =     90
```

Description

The result is the same, 90 pennies as shown below:



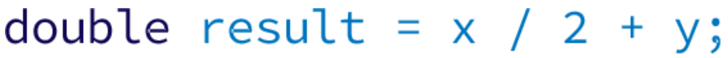
```
jshell> int pennies = 2 * 25 + 3 * 10 + 2 * 5;
pennies ==> 90

jshell> pennies = ( 2 * 25 ) + ( 3 * 10 ) + ( 2 * 5 );
pennies ==> 90
```

Description

It sometimes helps to use parentheses to clarify the order of calculations, but parentheses are essential when our desired order of evaluation is different from the rules of operator precedence. For example, to calculate the value of this formula:

$$\frac{x}{2+y}$$

we could write this code:



```
double result = x / 2 + y;
```

This would generate incorrect results because, according to the rules of precedence, *x/2* would be calculated first, then *y* would be added to the result of that division. In algebraic terms, the preceding statement is equivalent to:

$$\frac{x}{2} + y$$

To code the original formula correctly, we need to use parentheses to force the addition to occur before the division:

```
double result = x / ( 2 + y );
```

2.3.4 Programming Activity 1: Converting Inches to Centimeters

Now that we know how to define variables and constants and make calculations, let's put this all together by writing a program that converts inches into the equivalent centimeters.

Locate the *MetricLength.java* source file found in this chapter's Programming Activity 1 folder in the supplied code files. Copy the file to your computer.

Open the *MetricLength.java* source file. You'll notice that the class already contains some source code. Your job is to fill in the blanks.

When we write a program, we begin by considering these questions:

1. What data values does the program require?
    a. What data values do we know?
    b. What data values will change from one execution of the program to the next?
2. What processing (algorithm) do we need to implement?
3. What is the output?

The comments in the source file will guide you through the answers to these questions, and by doing so, you will complete the program. Search for five asterisks in a row (*****). This will position you to the places in the source code where you will add your code. The *MetricLength.java* source code is shown in **Example 2.7**. Sample output for a value of 5.2 inches is shown in **Figure 2.5**.

Example 2.7 Converting Inches to Centimeters

```java
/* MetricLength - converts inches to centimeters
   Anderson, Franceschi
*/

public class MetricLength
{
   public static void main( String [ ] args )
   {

      /***** 1. What data values do we know?
         We know that there are 2.54 centimeters in an inch.
         Declare a double constant named CM_PER_INCH.
         Assign CM_PER_INCH the value 2.54.
      */


      /***** 2. What other data does the program require?
         For this program, we require the number of inches.
         Declare a double variable named inches.
         Assign any desired value to this variable.
      */


      /***** 3. Calculation: convert inches to centimeters
         Declare a double variable named centimeters.
         Multiply inches by CM_PER_INCH
         and store the result in centimeters.
      */


      /***** 4. Output
         Write one or two statements that output
         the original inches and the equivalent centimeters.
         Try to match the sample output in Figure 2.5
      */



   }
}
```

Description

Figure 2.5 Sample Output for Programming Activity 1

DISCUSSION QUESTIONS

1. How do you know that your program results are correct?
2. If you change the inches data value, does your program still produce correct results?

2.3.5 Integer Division and Modulus

Division of a floating-point number by another floating-point number will result in a floating-point number; that is, a number with a fractional part. This is like the result you would receive using a calculator. For example, 12.6 / 2.4 will yield 5.25. Division with two integer operands, however, is performed in the Arithmetic Logic Unit (ALU), which can calculate only an integer result. Any fractional part is truncated; no rounding is performed. The remainder after division is available, however, as an integer, by taking the modulus (%) of the two integer operands. Thus, in Java, the integer division (/) operator will calculate the quotient of the division, whereas the modulus (%) operator will calculate the remainder of the division.

In **Example 2.8**, we have 113 pennies and we want to convert those pennies into quarters. We can find the number of quarters by dividing 113 by 25. The *int* variable *pennies* is assigned the value 113 at line 10. At line 12, the variable *quarters* is assigned the result of the integer division of *pennies* by the constant *PENNIES_PER_QUARTER*. Since the quotient of the division of 113 by 25 is 4, *quarters* will be assigned 4. At line 16, we use the modulus operator to assign to the variable *penniesLeftOver* the remainder of the division of *pennies* by *PENNIES_PER_QUARTER*. Since the remainder of the division of 113 by 25 is 13, 13 will be assigned to *penniesLeftOver*. Notice that integer division and modulus are independent calculations. We can perform a division without also calculating the modulus, and we can calculate the modulus without performing the division.

Example 2.8 How Integer Division and Modulus Work

```
 1 /* DivisionAndModulus Class
 2    Anderson, Franceschi
 3 */
 4
 5 public class DivisionAndModulus
 6 {
 7    public static void main( String [ ] args )
 8    {
 9      final int PENNIES_PER_QUARTER = 25;
10      int pennies = 113;
11
12      int quarters = pennies / PENNIES_PER_QUARTER;
13      System.out.println( "There are " + quarters + " quarters in "
14              + pennies + " pennies" );
15
16      int penniesLeftOver = pennies % PENNIES_PER_QUARTER;
17      System.out.println( "There are " + penniesLeftOver
18              + " pennies left over" );
19
20      final double MONTHS_PER_YEAR = 12;
21      double annualSalary = 50000.0;
22
23      double monthlySalary = annualSalary / MONTHS_PER_YEAR;
24      System.out.println( "The monthly salary is " + monthlySalary );
25    }
26 }
```

Description

At line 23, we divide a *double* by a *double;* therefore, a floating-point division will be performed by the floating-point unit (FPU), and the floating-point result will be assigned to the variable *monthlySalary*. **Figure 2.6** shows the output of the program.

Figure 2.6 Output of Example 2.8

Description

The modulus is a useful operator. It can be used to determine whether a number is even or odd, to control the number of data items that are written per line, to determine if one number is a factor of another, and for many other uses, as we will see later in this text.

CODE IN ACTION

To see arithmetic operators used in a program, look for this chapter's movie within the online resources. Click on this chapter's link to start the movie.

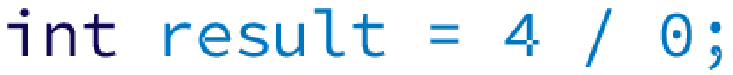
© Hemera Technologies/Photos.com/Thinkstock

with these end-of-chapter questions

2.6.2	Reading and Understanding Code Questions 7, 8, 9, 10, 11, 12, 13
2.6.3	Fill In the Code Questions 27, 29, 32
2.6.4	Identifying Errors in Code Question 35
2.6.6	Write a Short Program Question 44

2.3.6 Division by Zero

As we might expect, Java does not allow integer division by 0. If we include this statement in our program,



```
int result = 4 / 0;
```

the code will compile without errors, but at run time, when this statement is executed, the JVM will generate an exception and print an error message on the Java console:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

In most cases, this stops the program. After we discuss selection techniques, we will be able to avoid dividing by zero by first testing whether the divisor is zero before performing the division.

In contrast, floating-point division by zero does not generate an exception. If the dividend is not zero, the answer is *Infinity*. If both the dividend and divisor are zero, the answer is *NaN*, which stands for “Not a Number.”

Example 2.9 illustrates the three cases of dividing by zero. As we can see on the output shown in **Figure 2.7**, line 16 never executes. We can see from the last line in the figure:

Example 2.9 Results of Division by Zero

```
 1 /* DivisionByZero Class
 2    Anderson, Franceschi
 3 */
 4
 5 public class DivisionByZero
 6 {
 7    public static void main( String [ ] args )
 8    {
 9      double result1 = 4.3 / 0.0;
10      System.out.println( "The value of result1 is " + result1 );
11
12      double result2 = 0.0 / 0.0;
13      System.out.println( "The value of result2 is " + result2 );
14
15      int result3 = 4 / 0;
16      System.out.println( "The value of result3 is " + result3 );
17    }
18 }
```

Description

Figure 2.7 Output of Example 2.9

Description

```
at DivisionByZero.main(DivisionByZero.java.15)
```

that an exception is generated at line 15 and the program halts execution. Line 16 was not executed.

Although floating-point division by zero doesn't bring our program to a halt, it doesn't provide useful results either. It's a good practice to avoid dividing by zero in the first place.

2.3.7 Mixed-Type Arithmetic and Type Casting

So far, we've used a single data type in the expressions we've evaluated. But life isn't always like that. Calculations often involve data of different primitive types.

When calculations of mixed types are performed, lower-precision operands are converted, or **promoted**, to the type of the operand that has the higher precision.

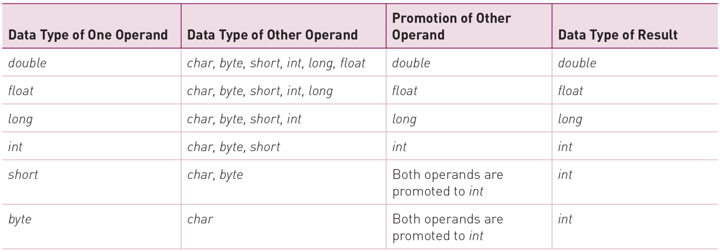
The promotions are performed using the *first* of these rules that fits the situation:

1. If either operand is a *double*, the other operand is converted to a *double*.

2. If either operand is a *float*, the other operand is converted to a *float*.
3. If either operand is a *long*, the other operand is converted to a *long*.
4. If either operand is an *int*, the other operand is promoted to an *int*.
5. If neither operand is a *double*, *float*, *long*, or an *int*, both operands are promoted to *int*.

Table 2.11 summarizes these rules of promotion.

Rules of Operand Promotion



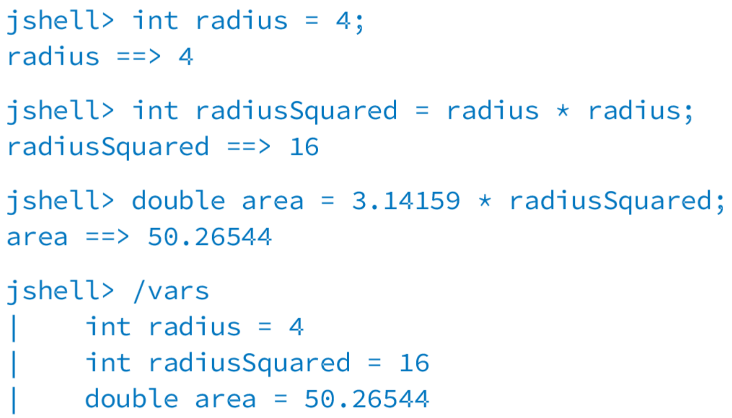
Data Type of One Operand	Data Type of Other Operand	Promotion of Other Operand	Data Type of Result
double	*char, byte, short, int, long, float*	*double*	*double*
float	*char, byte, short, int, long*	*float*	*float*
long	*char, byte, short, int*	*long*	*long*
int	*char, byte, short*	*int*	*int*
short	*char, byte*	Both operands are promoted to *int*	*int*
byte	*char*	Both operands are promoted to *int*	*int*

Description

This arithmetic promotion of operands is called **implicit type casting** because the compiler performs the promotions automatically, without our specifying that the conversions should be made. Note that the data type of any promoted variable is not permanently changed; its type remains the same after the calculation has been performed.

Table 2.11 shows many rules, but essentially, any arithmetic expression involving integers and floating-point numbers will evaluate to a floating-point number.

The code snippet below illustrates the rules of promotion. We calculate the area of a circle using a radius that is an *int* and 3.14159, which is a *double*. Our first calculation is to square the radius. Because *radius* is an *int*, the result is also an *int*. Then we multiply *radiusSquared* by 3.14159, a *double*. At this point, the *radiusSquared* value is promoted to a *double* and the resulting area is a *double*. Note that when we list our variables that *radiusSquared* is still an *int*.



```
jshell> int radius = 4;
radius ==> 4

jshell> int radiusSquared = radius * radius;
radiusSquared ==> 16

jshell> double area = 3.14159 * radiusSquared;
area ==> 50.26544

jshell> /vars
|    int radius = 4
|    int radiusSquared = 16
|    double area = 50.26544
```

Description

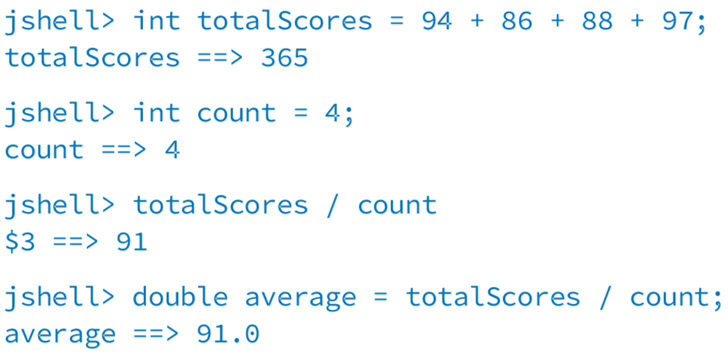
Sometimes, it’s useful to instruct the compiler specifically to convert the type of a variable. In this case, we use **explicit type casting**, which uses this syntax:

```
(dataType) ( expression )
```

The expression will be converted, or type cast, to the data type specified. The parentheses around *expression* are needed only when the expression consists of a calculation that we want to be performed before the type casting. If the expression consists of a single variable or value, the parentheses around *expression* are not needed.

Type casting is useful in calculating an average. The snippet below shows how to calculate an average test grade. The test scores are 94, 86, 88, and 97, making the combined total score 365. We expect the average to be 91.25.

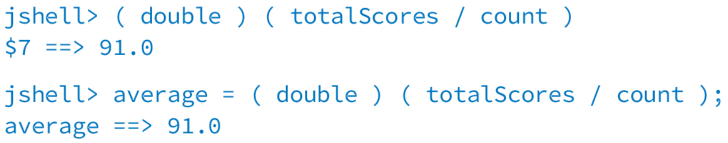
Our first attempt to calculate the average results in the wrong answer. Because *totalScores* and *count* are both integers, integer division is performed. This truncates any remainder, as shown by the value of the temporary variable $3 (91). When we then assign the integer result to the *double average*, the integer value is promoted to a *double*. The result, 91.0, appears to be accurate to one decimal place, but it is, in fact, incorrect.



```
jshell> int totalScores = 94 + 86 + 88 + 97;
totalScores ==> 365

jshell> int count = 4;
count ==> 4

jshell> totalScores / count
$3 ==> 91

jshell> double average = totalScores / count;
average ==> 91.0
```

Description

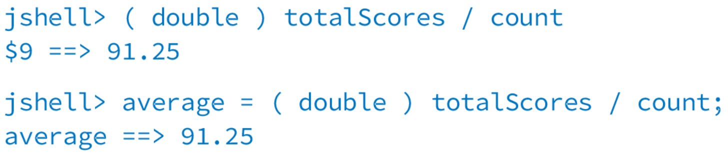
In our second attempt, we explicitly type cast the division to a double. But we are too late. The integer division is performed first (which we’ve seen is 91), then the result of the integer division is type cast to a double. Again, we receive the same incorrect and misleading result, 91.0.



```
jshell> ( double ) ( totalScores / count )
$7 ==> 91.0

jshell> average = ( double ) ( totalScores / count );
average ==> 91.0
```

Description

In our third attempt, we calculate the correct answer by type casting the *totalScores* variable to a *double*, then dividing. The result is that floating-point division is performed and we get the correct answer, 91.25. It doesn’t matter whether we type cast *totalScores* or *count* to a *double* before performing the division. Casting either to a *double* forces floating-point division. Casting has higher precedence than division.



```
jshell> ( double ) totalScores / count
$9 ==> 91.25

jshell> average = ( double ) totalScores / count;
average ==> 91.25
```

Description

CODE IN ACTION

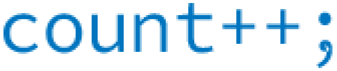
To see the calculation of an average using mixed data types, look for this chapter’s movie within the online resources. Click on this chapter’s link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

2.3.8 Shortcut Operators

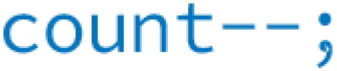
A common operation in programming is adding 1 to a number - (**incrementing**) or subtracting 1 from a number (**decrementing**). For example, if we want to count how many data items the user enters, every time we read another data item, we could add 1 to a count variable.

Because incrementing or decrementing a value is so common in programming, Java provides shortcut operators to do this: ++ and −−. (Note that there are no spaces between the two plus and minus signs.) The statement



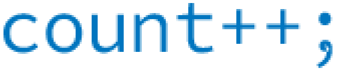
```
count++;
```

adds 1 to the value of *count*, and the statement



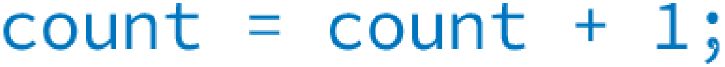
```
count--;
```

subtracts 1 from the value of *count*. Thus,



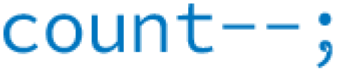
```
count++;
```

is equivalent to



```
count = count + 1;
```

and



```
count--;
```

is equivalent to

```
count = count - 1;
```

Both of these operators have **prefix** and **postfix** versions. The prefix versions precede the variable name (`++a` or `--a`) whereas the postfix versions follow the variable name (`a++` or `a--`). Both increment or decrement the variable. If they are used as a single, atomic statement (as in the preceding statements), there is no difference between the two versions. So

```
a++;
```

is functionally equivalent to

```
++a;
```

and

```
a--;
```

is functionally equivalent to

```
--a;
```

However, if they are used inside a more complex expression, then they differ as follows. The prefix versions increment or decrement the variable first, then the new value of the variable is used in evaluating the expression. The postfix versions increment or decrement the variable after the old value of the variable is used in the expression.

Example 2.10 illustrates this difference.

Example 2.10 Prefix and Postfix Increment Operators

```java
1  /* ShortcutOperators Class
2     Anderson, Franceschi
3  */
4
5  public class ShortcutOperators
6  {
7    public static void main( String [ ] args )
8    {
9      int a = 6;
10     int b = 2;
11
12     System.out.println( "At the beginning, a is " + a );
13     System.out.println( "Increment a with prefix notation: " + ++a );
14     System.out.println( "In the end, a is " + a );
15
16     System.out.println( "\nAt the beginning, b is " + b );
17     System.out.println( "Increment b with postfix notation: " + b++ );
18     System.out.println( "In the end, b is " + b );
19   }
20 }
```

Description

Lines 9 and 10 declare and initialize two *int* variables, *a* and *b*, to 6 and 2, respectively. In order to illustrate the effect of both the prefix and postfix increment operators, we output their original values at lines 12 and 16. At line 13, we use the prefix increment operator to increment *a* inside an output statement; *a* is incremented before the output statement is executed, resulting in the output statement using the value 7 for *a*. At line 17, we use the postfix increment operator to increment *b* inside an output statement; *b* is incremented after the output statement is executed, resulting in the output statement using the value 2 for *b*. Lines 14 and 18 simply output the values of *a* and *b* after the prefix and postfix operators were used at lines 13 and 17. **Figure 2.8** shows the output of this example.

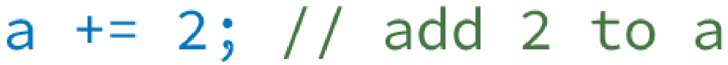
Figure 2.8 Output of Example 2.10

Description

Another set of shortcut operators simplify common calculations that change a single value. For example, the statement

```java
a = a + 2: // add 2 to a
```

can be simplified as



```
a += 2; // add 2 to a
```

The value added to the target variable can be a variable name or a larger expression.

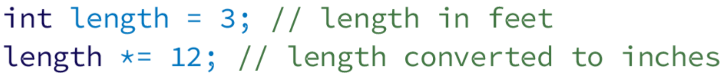
The shortcut addition operator (+=) is a single operator; there are no spaces between the 1 and the 5. Also, be careful not to reverse the order of the operators. For example, in the following statement, the operators are reversed, so the compiler interprets the statement as “assign positive 2 to a.”

COMMON ERROR TRAP

No spaces are allowed between the arithmetic operator (1) and the equal sign. Note also that the sequence is 15, not 51.

```
a =+ 2 ; // Incorrect! Assigns positive 2 to a
```

Java provides shortcut operators for each of the basic arithmetic operations: addition, subtraction, multiplication, division, and modulus. These operators are especially useful in performing repetitive calculations and in converting values from one scale to another. For example, to convert feet to inches, we multiply the number of feet by 12. So we can use the *5 shortcut operator:



```
int length = 3; // length in feet
length *= 12; // length converted to inches
```

Description

Converting from one scale to another is a common operation in programming. For example, earlier in the chapter we converted quarters, dimes, and nickels to pennies. We might also need to convert hours to seconds, feet to meters, or Fahrenheit temperatures to Celsius.

Example 2.11 demonstrates each of the shortcut arithmetic operators. The output is shown in **Figure 2.9**.

Example 2.11 Shortcut Arithmetic Operators

```java
/* Shortcut Arithmetic Operators
   Anderson, Franceschi
*/

public class ShortcutArithmeticOperators
{
  public static void main( String [ ] args )
  {
    int a = 5;
    System.out.println( "a is " + a );

    a += 10;      // a = a + 10;
    System.out.println( "\nAfter a += 10; a is " + a );

    a -= 3;       // a = a - 3;
    System.out.println( "\nAfter a -= 3; a is " + a );

    a *= 2;       // a = a * 2;
    System.out.println( "\nAfter a *= 2; a is " + a );

    a /= 6;       // a = a / 6;
    System.out.println( "\nAfter a /= 6; a is " + a );

    a %= 3;       // a = a % 3;
    System.out.println( "\nAfter a %= 3; a is " + a );
  }
}
```

Description

Figure 2.9 Output of Example 2.11

Description

Table 2.12 summarizes the shortcut operators, and **Table 2.13** shows where the shortcut operators fit into the order of operator precedence.

Shortcut Operators

Shortcut Operator	**Example**	**Equivalent Statement**
+ +	a + +; or + + a;	a = a + 1;
– –	a– –; or – –a;	a = a – 1;
+ =	a + = 3;	a = a + 3;

– =	a – = 10;	a = a – 10;
* =	a *= 4;	a = a * 4;
/ =	a / = 7;	a = a / 7;
% =	a % = 10;	a = a % 10;

Description

Order of Operator Precedence

Operator Hierarchy	**Order of Same-Statement Evaluation**	**Operation**
( )	left to right	parentheses for explicit grouping
+ +, – –	**right to left**	**shortcut postincrement**
+ +, – –	**right to left**	**shortcut preincrement**
+, /, %	left to right	multiplication, division, modulus
+, –	left to right	addition or *String* concatenation, subtraction
=, + =, – =, * =, / =, % =	right to left	assignment operator and **shortcut assignment operators**

Description

with these end-of-chapter questions

2.6.1	Multiple Choice Exercises Question 3
2.6.2	Reading and Understanding Code Questions 14, 15, 16, 17, 18, 19, 20, 21, 22
2.6.3	Fill In the Code

	Questions 28, 30, 31
2.6.4	Identifying Errors in Code Questions 36, 37
2.6.5	Debugging Area Questions 42, 43
2.6.6	Write a Short Program Question 45
2.6.8	Technical Writing Question 51

2.4 Programming Activity 2: Temperature Conversion

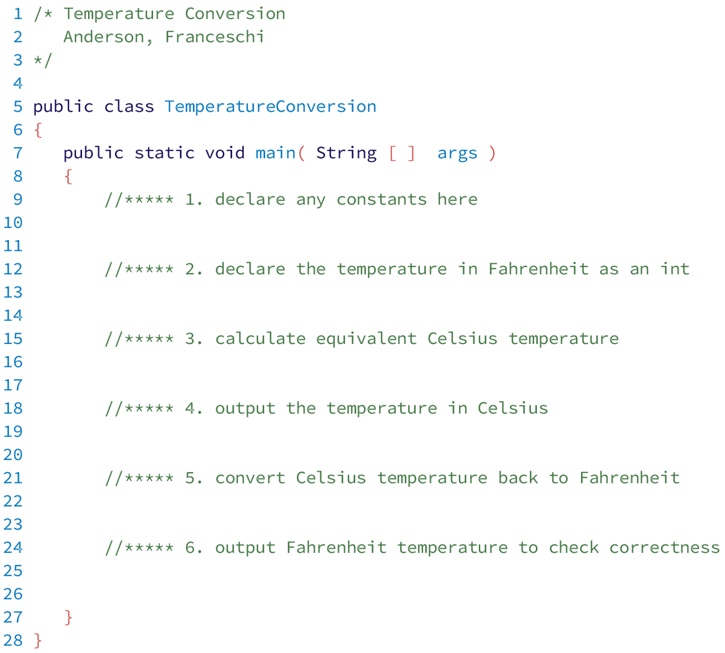
For this Programming Activity, you will write a program to convert a temperature in Fahrenheit to Celsius. The conversion formula is the following:

$$T_c = 5/9\ (T_f - 32)$$

where T_c is the temperature in Celsius and T_f is the temperature in Fahrenheit, and 32 is the freezing point of water.

Locate the *TemperatureConversion.java* source file found in this chapter's Programming Activity 2 folder in the supplied code files. Copy the file to your computer. The source code is shown in **Example 2.12**.

Example 2.12 TemperatureConversion.java



```java
/* Temperature Conversion
   Anderson, Franceschi
*/

public class TemperatureConversion
{
   public static void main( String [ ]  args )
   {
       //***** 1. declare any constants here


       //***** 2. declare the temperature in Fahrenheit as an int


       //***** 3. calculate equivalent Celsius temperature


       //***** 4. output the temperature in Celsius


       //***** 5. convert Celsius temperature back to Fahrenheit


       //***** 6. output Fahrenheit temperature to check correctness


   }
}
```

Description

Open the *TemperatureConversion.java* source file. You'll notice that the class already contains a class name and the *main* method. Your job is to fill in the blanks.

To verify that your code produces the correct output, add code to convert your calculated Celsius temperature back to Fahrenheit and compare that value to the original Fahrenheit temperature. The formula for converting Celsius to Fahrenheit is:

$$T_f = 9/5 * T_c + 32$$

Before writing this program, you need to design a plan of attack. Ask yourself:

- What data do I need to define?
- What calculations should I make?
- What is the output of the program?

- How do I select data values so they will provide good test data for my code?

Choose any input value for the Fahrenheit temperature. After you write the program, try changing the original temperature value, recompiling and rerunning the program to verify that the temperature conversion works for multiple input values.

DISCUSSION QUESTIONS

1. How did you change the expression 5 / 9 so that the value was not 0?
2. What constant(s) did you define?
3. What data type did you use for the Celsius temperature? Why?

© Lois GoBe/Shutterstock.

CHAPTER REVIEW

2.5 Chapter Summary

- Java programs consist of at least one class.
- Identifiers are symbolic names for classes, methods, and data. Identifiers should start with a letter and may contain any combination of letters and digits, but no spaces. The length of an identifier is essentially unlimited. Identifier names are case sensitive.
- Java's reserved words cannot be used as identifiers.
- The basic building block of a Java program is the statement. A statement is terminated with a semicolon and can span several lines.
- Any amount of white space is permitted between identifiers, Java keywords, operands, operators, and literals. White space characters are the space, tab, newline, and carriage return.
- A block, which consists of 0, 1, or more statements, starts with a left curly brace and ends with a right curly brace. Blocks can be used anywhere in the program that a statement is legal.
- Comments are ignored by the compiler. Block comments are delineated by /* and */. Line comments start with // and continue to the end of the line.
- Java supports eight primitive data types: *double*, *float*, *long*,

int, *short*, *byte*, *char*, and *boolean*.

- The jshell utility, part of the JDK, is a tool that allows us to type code and see the immediate result without compiling.
- Variables must be declared before they are used. Declaring a variable is specifying the data item's identifier and data type. The syntax for declaring a variable is: `datatype identifier1, identifier2, . . .;`
- Begin variable names with a lowercase letter. If the variable name consists of more than one word, begin each word after the first with a capital letter. Do not put spaces between words.
- An integer data type is one that evaluates to a positive or negative whole number. Java recognizes four integer data types: *int*, *short*, *long*, and *byte*.
- Floating-point data types store numbers with fractional parts. Java supports two floating-point data types: the single-precision type *float*, and the double-precision type *double*.
- The *char* data type stores one Unicode character. Because Unicode characters are encoded as unsigned numbers using 16 bits, a *char* variable is stored in two bytes of memory.
- The *boolean* data type can store only two values, which are expressed using the Java reserved words *true* and *false*.
- The assignment operator (5) is used to give a value to a variable.
- To assign an initial value to a variable, use this syntax when declaring the variable:

```
dataType variable1 = initialValue1;
```

- Literals can be used to assign initial values or to reassign the value of a variable.
- Constants are data items whose value, once assigned, cannot be changed. Data items that we know should not change throughout the execution of a program should be declared as a constant, using this syntax:

```
final dataType CONSTANT_IDENTIFIER = initialValue;
```

Description

- Constant identifiers, by convention, are composed of all capital letters with underscores separating words.
- An expression consists of operators and operands that evaluate to a single value.
- The value of an expression can be assigned to a variable or constant, which must be a data type compatible with the value of the expression and cannot be a constant that has been assigned a value already.
- Java provides binary operators for addition, subtraction, multiplication, division, and modulus.
- Calculation of the value of expressions follows the rules of operator precedence.
- Integer division truncates any fractional part of the quotient.
- When an arithmetic operator is invoked with operands that are of different primitive types, the compiler temporarily converts, or promotes, one or both of the operands.
- An expression or a variable can be temporarily cast to a different data type using this syntax:

```
(dataType) ( expression )
```

Description

- Shortcut operators + + and – – simplify incrementing or decrementing a value by 1. The prefix versions precede the variable name and increment or decrement the variable, then use its new value in evaluation of the expression. The postfix versions follow the variable name and increment or decrement the variable after using the old value in the expression.
- Java provides shortcut operators for each of the basic arithmetic operations: addition, subtraction, multiplication, division, and modulus.

2.6 Exercises, Problems, and Projects

2.6.1 Multiple Choice Exercises

1. What is the valid way to declare an integer variable named *a*? (Check all that apply.)
    - `int a;`
    - `a int ;`
    - `integer a;`
5. Which of the following identifiers are valid?
    - `a`

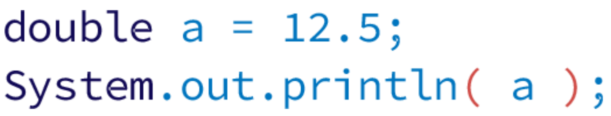
- `sales`
- `sales&profit`
- `int`
- `inter`
- `doubleSales`
- `TAX_RATE`
- `1stLetterChar`
- `char`

15. Given three declared and initialized *int* variables *a*, *b*, and *c*, which of the following statements are valid?

    - `a = b;`
    - `a = 67;`
    - `b = 8.7;`
    - `a + b = 8;`
    - `a * b = 12;`
    - `c = a – b;`
    - `c = a / 2.3;`
    - `boolean t = a;`
    - `a /= 4;`
    - `a += c;`

2.6.2 Reading and Understanding Code

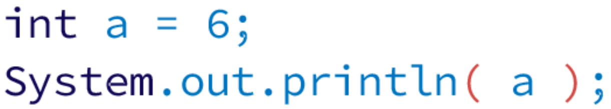
1. What is the output of this code sequence?



```
double a = 12.5;
System.out.println( a );
```

Description

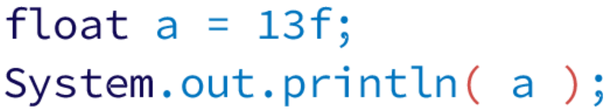
2. What is the output of this code sequence?



```
int a = 6;
System.out.println( a );
```

Description

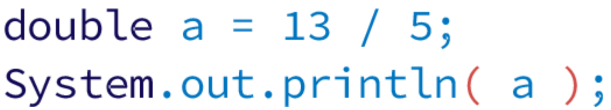
3. What is the output of this code sequence?



```
float a = 13f;
System.out.println( a );
```

Description

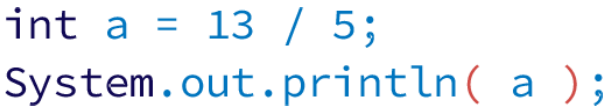
4. What is the output of this code sequence?



```
double a = 13 / 5;
System.out.println( a );
```

Description

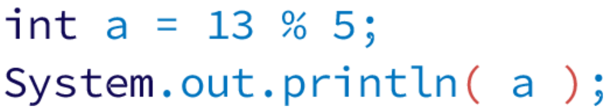
5. What is the output of this code sequence?



```
int a = 13 / 5;
System.out.println( a );
```

Description

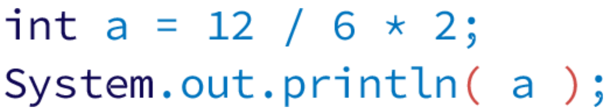
6. What is the output of this code sequence?



```
int a = 13 % 5;
System.out.println( a );
```

Description

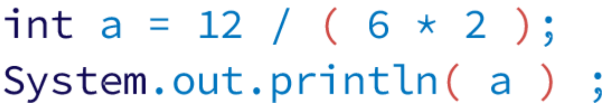
7. What is the output of this code sequence?



```
int a = 12 / 6 * 2;
System.out.println( a );
```

Description

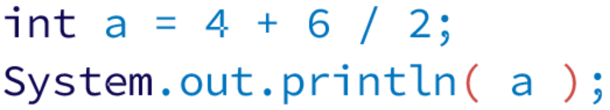
8. What is the output of this code sequence?



```
int a = 12 / ( 6 * 2 );
System.out.println( a ) ;
```

Description

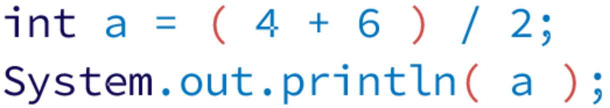
9. What is the output of this code sequence?



```
int a = 4 + 6 / 2;
System.out.println( a );
```

Description

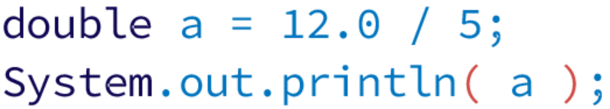
10. What is the output of this code sequence?



```
int a = ( 4 + 6 ) / 2;
System.out.println( a );
```

Description

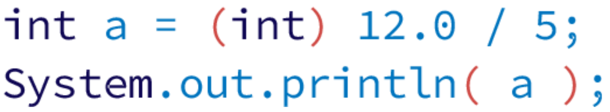
11. What is the output of this code sequence?



```
double a = 12.0 / 5;
System.out.println( a );
```

Description

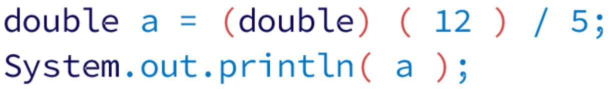
12. What is the output of this code sequence?



```
int a = (int) 12.0 / 5;
System.out.println( a );
```

Description

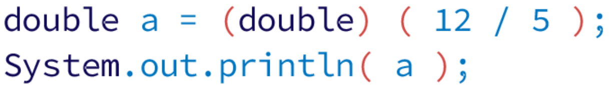
13. What is the output of this code sequence?



```
double a = (double) ( 12 ) / 5;
System.out.println( a );
```

Description

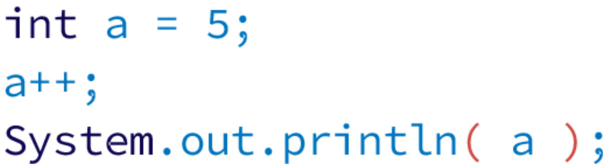
14. What is the output of this code sequence?



```
double a = (double) ( 12 / 5 );
System.out.println( a );
```

Description

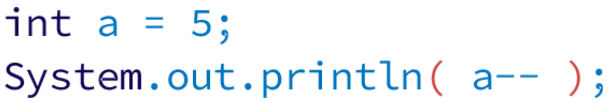
15. What is the output of this code sequence?



```
int a = 5;
a++;
System.out.println( a );
```

Description

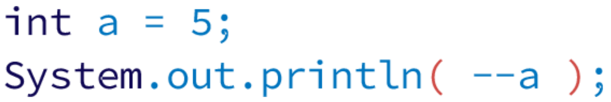
16. What is the output of this code sequence?



```
int a = 5;
System.out.println( a-- );
```

Description

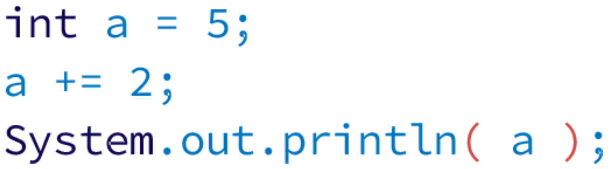
17. What is the output of this code sequence?



```
int a = 5;
System.out.println( --a );
```

Description

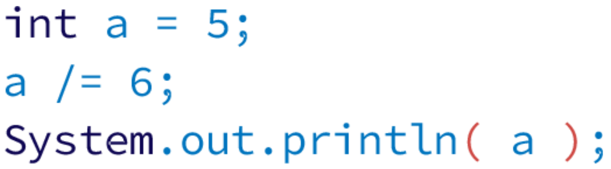
18. What is the output of this code sequence?



```
int a = 5;
a += 2;
System.out.println( a );
```

Description

19. What is the output of this code sequence?

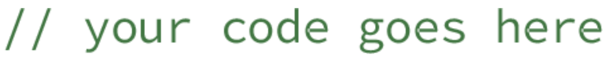


```
int a = 5;
a /= 6;
System.out.println( a );
```

Description

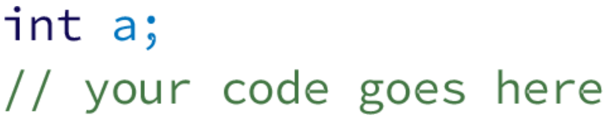
2.6.3 Fill In the Code

1. Write the code to declare a *float* variable named *a* and assign *a* the value 34.2.



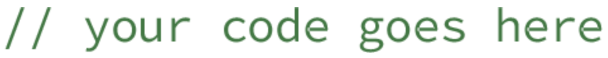
```
// your code goes here
```

2. Write the code to assign the value 10 to an *int* variable named *a*.



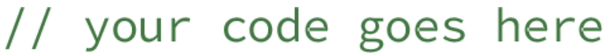
```
int a;
// your code goes here
```

3. Write the code to declare a *boolean* variable named *a* and assign *a* the value *false*.



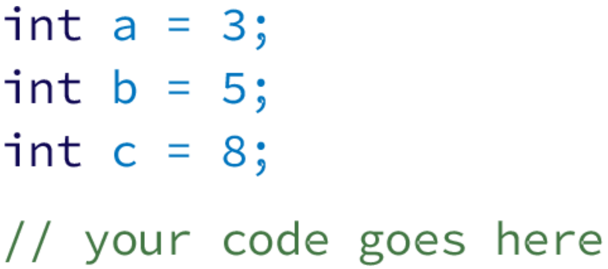
```
// your code goes here
```

4. Write the code to declare a *char* variable named *a* and assign *a* the character B.



```
// your code goes here
```

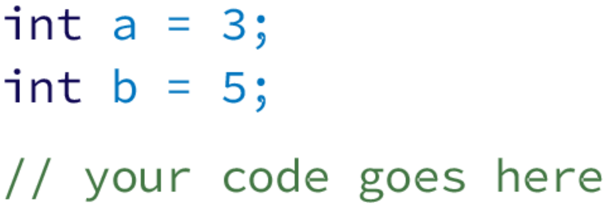
5. Write the code to calculate the total of three *int* variables *a*, *b*, and *c* and print the result.



```
int a = 3;
int b = 5;
int c = 8;

// your code goes here
```

Description

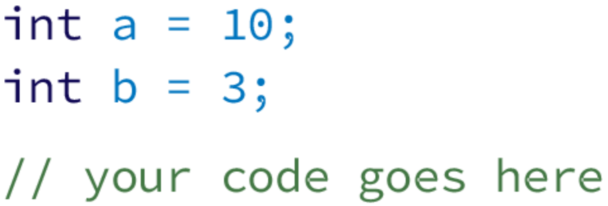
6. Write the code to calculate the average of two *int* variables *a* and *b* and print the result. The average should be printed as a floating-point number.



```
int a = 3;
int b = 5;

// your code goes here
```

Description

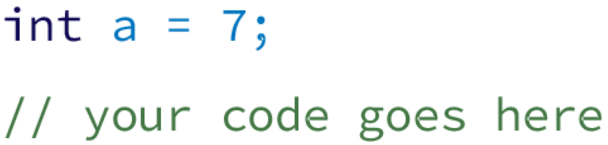
7. Write the code to calculate and print the remainder of the division of two *int* variables with the values 10 and 3 (the value printed will be 1).



```
int a = 10;
int b = 3;

// your code goes here
```

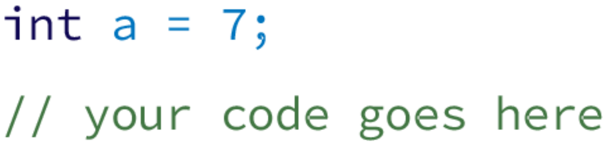
Description

8. This code increases the value of a variable *a* by 1, using the shortcut increment operator.



```
int a = 7;

// your code goes here
```

9. This code multiplies the value of a variable *a* by 3, using a shortcut operator.



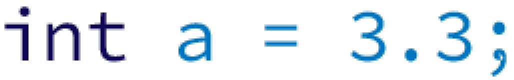
```
int a = 7;

// your code goes here
```

10. Assume that we have already declared and initialized two *int* variables, *a* and *b*. Convert the following sentences to legal Java expressions and statements.

    - b gets a plus 3 minus 7
    - b gets a times 4
    - a gets b times b
    - a gets b times 3 times 5
    - b gets the quotient of the division of a by 2
    - b gets the remainder of the division of a by 3

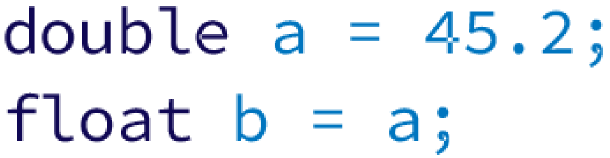
2.6.4 Identifying Errors in Code

1. Where is the error in this code sequence?



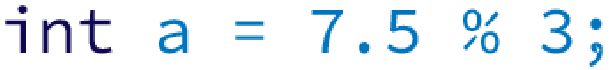
```
int a = 3.3;
```

2. Where is the error in this code sequence?



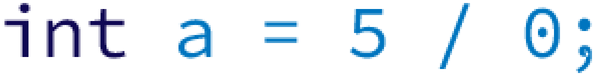
```
double a = 45.2;
float b = a;
```

3. Where is the error in this code sequence?



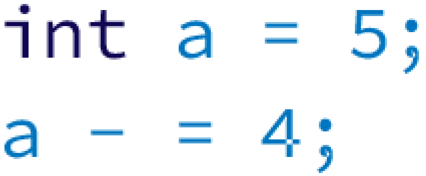
```
int a = 7.5 % 3;
```

4. What would happen when this code sequence is compiled and executed?



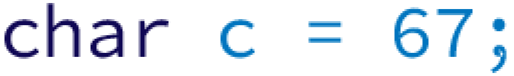
```
int a = 5 / 0;
```

5. Where is the error in this code sequence?



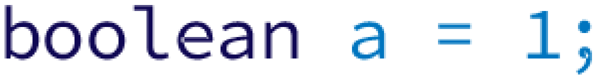
```
int a = 5;
a - = 4;
```

6. Is there an error in this code sequence? Explain.



```
char c = 67;
```

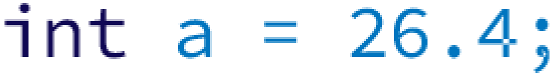
7. Is there an error in this code sequence? Explain.



```
boolean a = 1;
```

2.6.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

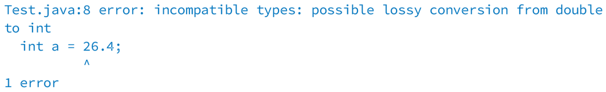
1. You coded the following on line 8 of class *Test.java*:



```
int a = 26.4;
```

Description

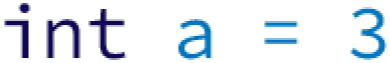
When you compile, you get the following message:



```
Test.java:8 error: incompatible types: possible lossy conversion from double
to int
  int a = 26.4;
          ^
1 error
```

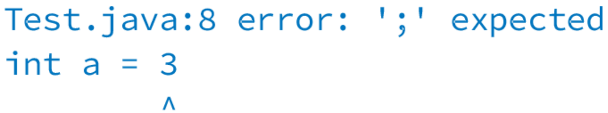
Explain what the problem is and how to fix it.

2. You coded the following on line 8 of class *Test.java*:



```
int a = 3
```

When you compile, you get the following message:

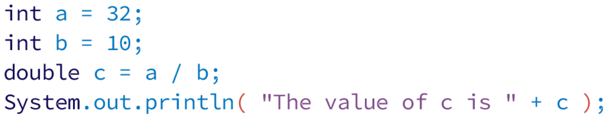


```
Test.java:8 error: ';' expected
int a = 3
         ^
```

Description

Explain what the problem is and how to fix it.

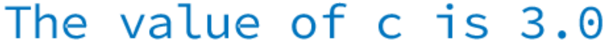
3. You coded the following in class *Test.java*:



```
int a = 32;
int b = 10;
double c = a / b;
System.out.println( "The value of c is " + c );
```

Description

The code compiles properly and runs, but the result is not what you expected. The output is



```
The value of c is 3.0
```

You expected the value of *c* to be 3.2. Explain what the problem is and how to fix it.

4. You coded the following in class *Test.java*:

```
int a = 5;
a =+ 3;
System.out.println( "The value of a is " + a );
```

Description

The code compiles properly and runs, but the result is not what you expected. The output is

```
The value of a is 3
```

You expected the value of *a* to be 8. Explain what the problem is and how to fix it.

2.6.6 Write a Short Program

1. Write a program that calculates and outputs the square of each integer from 1 to 9.
2. Write a program that calculates and outputs the average of integers 1, 7, 9, and 34.
3. Write a program that outputs the following:

```
****
```

2.6.7 Programming Projects

1. Write a program, not using a text block, that prints the letter X composed of asterisks (*). Your output should look like this:

```
*   *
 * *
  *
 * *
*   *
```

Description

2. Write a program, using a text block, that prints the letter X composed of asterisks (*). Your output should look like this:

```
*   *
 * *
  *
 * *
*   *
```

Description

3. Write a program that converts 10, 50, and 100 kilograms to pounds (1 lb = 0.454 kg).
4. Write a program that converts 2, 5, and 10 inches to millimeters (1 inch = 25.4 mm).
5. Write a program to compute and output the circumference of a circle having a radius of 3.2 inches.

6. Write a program that outputs this *String* :
   $a^3 + b^3 = c^3$

Description

2.6.8 Technical Writing

1. Some programmers like to write code that is as compact as possible, for instance, using the increment (or decrement) operator in the middle of another statement. Typically, these programmers use very few comments in their programs. Discuss whether this is a good idea, keeping in mind that a program “lives” through a certain period of time.
2. Compare the following data types for integer numbers: *int*, *short*, and *long*. Discuss their representation in binary, how much space they take in memory, and the purpose of having these data types available to programmers.

© Lois GoBe/Shutterstock.

CHAPTER 3

Object-Oriented Programming, Part 1: Using Classes

CHAPTER CONTENTS

Introduction
3.1 Class Basics and Benefits
3.2 Creating Objects Using Constructors
3.3 Calling Methods
3.4 Using Object References
3.5 More on the *String* Class
3.6 Programming Activity 1: Calling Methods
3.7 The Java Class Library
3.8 Formatting Output with the *DecimalFormat* Class
3.9 Generating Random Numbers with the *Random* Class
3.10 Input from the Console Using the *Scanner* Class
3.11 Calling *Static* Methods and Using *Static* Class Variables
3.12 Using *System.in* and *System.out*
3.13 The *Math* Class
3.14 Formatting Output with the *NumberFormat* Class
3.15 The *Integer*, *Double*, *Character*, and Other Wrapper Classes

3.16 Programming Activity 2: Using Predefined Classes
3.17 Chapter Summary
3.18 Exercises, Problems, and Projects

- 3.18.1 Multiple Choice Exercises
- 3.18.2 Reading and Understanding Code
- 3.18.3 Fill In the Code
- 3.18.4 Identifying Errors in Code
- 3.18.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
- 3.18.6 Write a Short Program
- 3.18.7 Programming Projects
- 3.18.8 Technical Writing
- 3.18.9 Group Project

Introduction

Writing computer programs that use classes and objects is called **object-oriented programming**, or **OOP**. Every Java program consists of at least one class.

In this chapter, we'll introduce object-oriented programming as a way to use classes that have already been written. Classes provide services to the program. These services might include writing a message to the program's user, popping up a dialog box, performing some mathematical calculations, formatting numbers, drawing shapes in a window, or many other basic tasks that add a more professional look to even simple programs. The program that uses a class is called the **client** of the class. Thus, in this chapter we will be writing client programs.

One benefit of using a prewritten class is that we don't need to write the code ourselves; it has already been written and tested for us. This means that we can write our client programs more quickly. In other words, we shorten the development time of the program. Using prewritten and pretested classes provides other benefits as well, including more reliable programs with fewer errors.

In this chapter, we'll explore how using prewritten classes can add functionality to our programs.

3.1 Class Basics and Benefits

In Java, classes are composed of data and operations—or functions—that operate on the data. Objects of a class are created using the class as a template, or guide. Think of the class as a generic description, and an object as a specific item of that class. Or you can think of a class as a cookie cutter; the objects of that class are the cookies made with the cookie cutter. For example, a *Student* class might have the following data: name, year, and grade point average. All students have these three data items. We can create an object of the *Student* class by specifying an identifier for the object (for example, *student1*) along with a name, year, and grade point average for a particular student (for example, *Maria Gonzales, Sophomore, 3.5*). The identifier of the object is called the **object reference**. Creating an object of a class is called **instantiating an object**, and the object is called an **instance of the class**. Many objects can be instantiated from one class. There can be many instances of the *Student* class, that is, many *Student* objects can be instantiated from the *Student* class. For example, we could create a second object of the *Student* class, *student2*, with its data as *Mike Smith, Junior, 3.0*.

The data associated with an object of a class are called **instance variables**, or **fields**, and can be variables and constants of any primitive data type (*byte, short, int, long, float, double, char*, and *boolean*), or they can be objects of a class.

The operations for a class, called **methods**, set the values of the data, retrieve the current values of the data, and perform other class-related functions on the data. For example, the *Student* class would provide methods to set the values of the name, year, and grade point average; retrieve the current values of the name, year, and grade point average; and perhaps promote a student to the next year. Invoking a method on an object is called **calling the method**. With a few exceptions, only the methods of a class can directly access or change the instance variables of an object. Programs using the class, such as clients, must call the methods to set or retrieve the values of the instance variables. Together, the fields and methods of a class are called its **members**.

In essence, a class is a new data type, which is created by combining items of Java primitive data types and objects of classes. Just as the primitive data types can be manipulated using arithmetic operators (+, –, *, /, and %), an object can be manipulated by calling the methods of its class.

The data of a class are typically declared to be *private*. In this case, only the methods of the class can change the data of an object, and thus the methods provide a protective layer around the data. In other words, the class **encapsulates** the data, and the methods of the class provide the only interface to set or change the data values from outside the class. The benefit from this encapsulation is that the class methods ensure that only valid values are assigned to an object. For example, a method to set a student's grade point average would accept values only between 0.0 and 4.0.

Let's look at another example of a class. The *String* class, an existing class of the Java library, stores a sequence of characters. An object of the *String* class, *firstName*, could be instantiated with the data value *Jane*. Another object of this class, *lastName*, might be instantiated with the data value *Jones*. Methods of the *String* class provide functionality related to a sequence of characters, such as retrieving the number of characters, converting the characters to uppercase, retrieving a character given its position, retrieving the position of a given character, and so on. Depending on the class, other methods could ensure that only valid values are set for the instance variables of the class. For example, if a class includes an instance variable named *price* that stores the price of an object, methods will ensure that the price is never negative.

Notice that the class names *Student* and *String* begin with a capital letter, and the object names, *student1*, *firstName* and *lastName*, start with a lowercase letter. By convention, class names start with a capital letter. Object names, instance variables, and method names conventionally start with a lowercase letter. Internal words start with a capital letter in class names, object names, variables, and methods.

SOFTWARE ENGINEERING TIP

By convention, class names in Java start with a capital letter. Method names, instance variables, and object names start with a lowercase letter. In all of these names, embedded words begin with a capital letter.

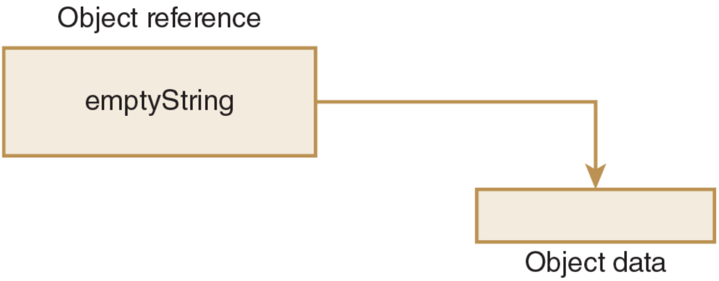
There are many benefits to using classes in a program. Some of the most important benefits include reusability (not only in the current program but also in other programs), encapsulation, and reliability.

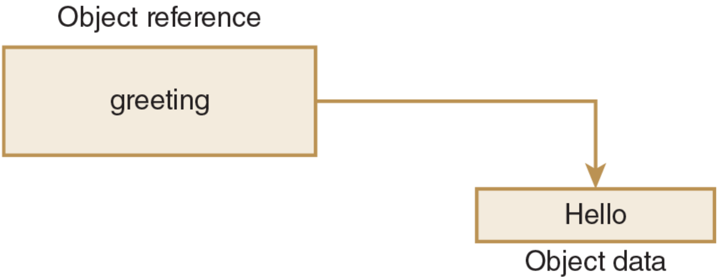
A well-written class can be reused in many programs. For example, the *String* class could be used in a program involving country and city names, login name and password, and many more applications that use words. Reusing code is much faster than writing and testing new code. As an added bonus, reusing a tested and debugged class in another program makes the program more reliable.

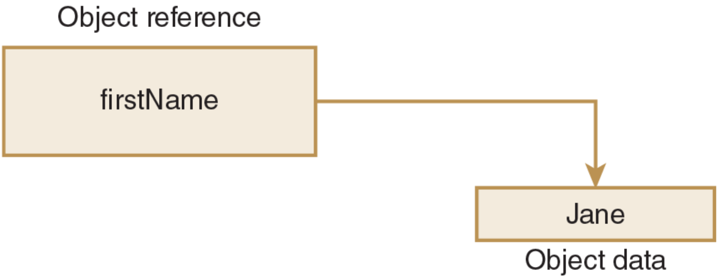
Further, encapsulation of a class's data and methods helps to isolate operations on the data. This makes it easier to track the source of a bug. For example, when a bug is discovered in an object of the *Student class*, then we know to look for the problem in the methods of the *Student class*, because no other code in our program can directly change the data in a *Student object*.

We do not need to know the implementation details of a class in order to use it in our program. The beauty of object orientation is that we don't need to know the implementation of the class; all we need to know is the class **application programming interface (API)**, that is, how to instantiate objects and how to call the methods of the class.

```java
/* A Demonstration of Using Constructors
   Anderson, Franceschi
*/

public class Constructors
{
  public static void main( String [ ] args )
  {
    String emptyString;
    emptyString = new String( );

    String greeting;
    greeting = new String( "Hello" );

    String firstName = new String( "Jane" );
  }
}
```


Object reference
emptyString
Object data


Object reference
greeting
Hello
Object data


Object reference
firstName
Jane
Object data

3.2 Creating Objects Using Constructors

A class describes a generic template for creating, or instantiating, objects. In fact, an object must be instantiated before it can be used. To understand how to instantiate an object of a class and how to call methods of the class, we must know the API of a class, which the creators of the class make public. **Table 3.1** shows part of the API of the *String* class.

String Class Constructors

String Class Constructor Summary

`String( String str )`	allocates a *String* object with the value of *str*, which can be a *String* object or a *String* literal
`String( )`	allocates an empty *String* object

Description

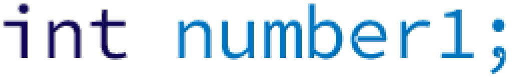
Instantiating an object consists of defining an object reference —which will hold the address of the object in memory—and calling a special method of the class called a **constructor**, which has the same name as the class. The job of the constructor is to assign initial values to the data of the class.

Example 3.1 illustrates how to instantiate objects of the *String* class.

Example 3.1 Demonstrating Constructors

Description

Declaring an object reference is very much like declaring a variable of a primitive type; we specify the data type and an identifier. For example, to declare an integer variable named *number1*, we provide the data type (*int*) and the identifier (*number1*), as follows:



```
int number1;
```

One notable difference in declaring an object reference is that its data type is the class name, not a primitive data type. Here is the syntax for declaring an object reference:

```
ClassName objectReference1, objectReference2, . . .;
```

In Example 3.1, lines 9, 12, and 15 declare object references for a *String* object. *String*, the class name, is the data type, and *emptyString, greeting*, and *firstName* are the object references.

Object references can refer to **any** object of its class. For example, *String* object references can point to any *String* object, but a *String* object reference cannot point to objects of other classes, such as a *Student* object.

Once an object reference has been declared, we instantiate the object using the following syntax:

```
objectReference = new ClassName( argument list );
```

Description

This calls a constructor of the class to initialize the data. The **argument list** consists of a comma-separated list of initial data values to assign to the object. Classes often provide multiple constructors with different argument lists. Depending on which constructor we call, we can accept default values for the data or specify initial values for the data. When we instantiate an object, our argument list—that is, the number of arguments and their data types—must match one of the constructors' argument lists.

Two constructors of the *String* class are shown in Table 3.1. Note that the API of a constructor does not have a return value; nevertheless, constructors implicitly return an object reference to the newly instantiated object. The first constructor, *String( )*, is called the **default constructor**, because its **argument list is empty**. This constructor assigns default values to all data in the object. Thus, in line 10 of Example 3.1, which uses the default constructor, the data for the *emptyString* object is set to the default value for the *String* class, which is an empty character sequence.

We see from Table 3.1 that the second constructor for the *String* class, *String*(*String original*), takes one argument, which evaluates to a *String*.

Lines 13 and 15 of Example 3.1 instantiate *String* objects using the second constructor. In line 13, the argument list tells the constructor to give the value *Hello* to *greeting*. Note that no data types are given when calling the constructors, only the initial values for the data. The data types of the arguments are specified in the API so that the client of the class knows what data types the constructor is expecting for its arguments.

Line 15 also illustrates that we can combine the declaration of the object reference and instantiation of the object in a single statement.

When an object is instantiated, the JVM allocates memory to the new object. The object reference is assigned an address that the JVM uses to find that object in memory. **Figure 3.1** shows the three objects instantiated in Example 3.1.

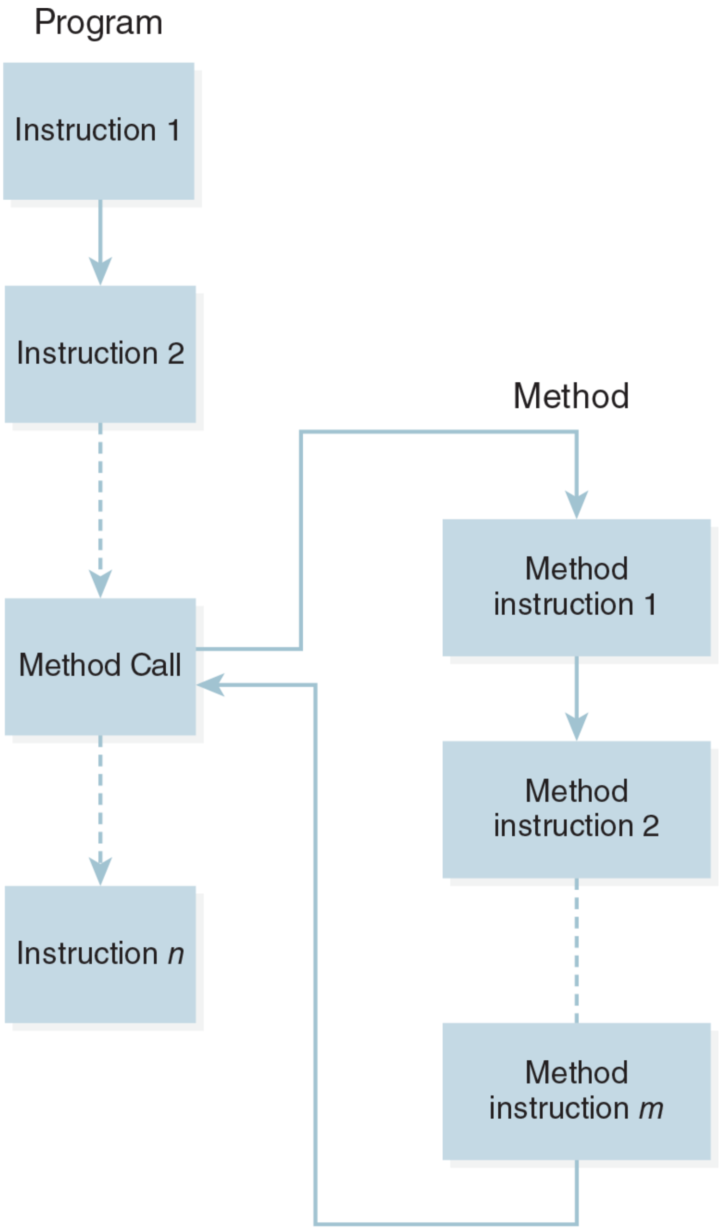
Figure 3.1 Three *String* Objects after Instantiation

Description

It’s important to understand that an object reference and the object data are different: The object reference represents a memory location. Notice in Figure 3.1 that the object references, *emptyString*, *greeting*, and *firstName*, point to the locations of the object data.

COMMON ERROR TRAP

Do not forget to instantiate all objects that your program needs. Objects must be instantiated before they can be used.


Program
Instruction 1
Instruction 2
Method Call
Instruction *n*
Method
Method instruction 1
Method instruction 2
Method instruction *m*

```
There are 12 characters
The index of M is 5
At index 3, we have a d
sub5 is Morning
In caps: GOOD MORNING
greeting is still Good Morning
```

3.3 Calling Methods

Once an object is instantiated, we can use the object by calling its methods. As we mentioned earlier, the authors of classes publish their API so that their clients know what methods are available and how to call those methods.

Figure 3.2 illustrates how calling a class method alters the flow of control in our program. When this program starts running, the JVM executes instruction 1, then instruction 2, then it encounters a method call.

Figure 3.2 Flow of Control of a Method Call

Description

At that point, the JVM **transfers control to the method** and starts executing instructions in the method. When the method finishes executing, the JVM transfers control back to the program immediately after the point the method was called and continues executing instructions in the program.

A class API consists of the method names, their return values, and their parameter lists. **Table 3.2** lists several methods of the *String* class. The parameter list for a method indicates the order and number of arguments to send to the method, along with the data type of each argument. Each item in the parameter list consists of a data type and a name. Formally, we use the term parameter when we talk about the method's API, and we use the term argument to describe a value passed to the method when we call it. Often, these two terms are used interchangeably. The arguments sent to the method can be literals, constants, variables, or any expression that evaluates to the data type specified in the API of the method. For example, the API in Table 3.2 shows that the *indexOf* method takes one argument, which must evaluate to a character value. It also shows that the *charAt* method takes one argument, which must evaluate to an integer value.

String Methods

String Class Method Summary

Return value	Method name and argument list
`int`	`length( )` returns the length of the *String*
`String`	`toUpperCase( )` returns a copy of the *String* with all letters in uppercase
`String`	`toLowerCase( )` returns a copy of the *String* with all letters in lowercase
`char`	`charAt( int index )` returns the character at the position specified by *index*
`int`	`indexOf( String searchString )` returns the index of the beginning of the first occurrence of *searchString* or –1 if *searchString* is not found
`int`	`indexOf( char searchChar )` returns the index of the first occurrence of *searchChar* in the *String* or –1 if *searchChar* is not found
`String`	`substring( int startIndex, int endIndex )` returns a substring of the *String* object beginning at the character at index *startIndex* and ending at the character at index *endIndex – 1*

`String`	`substring( int startIndex )` returns a substring of the *String* object beginning at the character at index *startIndex* and continuing to the end of the *String*
`boolean`	`isBlank( )` returns *true* if this *String* is empty or only contains white space character (including \t or \n)
`String`	`strip( )` returns a copy of this *String* with all the leading and trailing white characters removed (including \t or \n)

Description

A method may or may not return a value, as indicated by a data type, class type, or the keyword **void** in front of the method name. If the method returns a value, then the data type or class type of its **return value** will precede the method's name. For instance, in Table 3.2, the *indexOf* method returns a character value. The *charAt* method returns an integer value. The call to a **value-returning method** can be used in an expression. When the method finishes executing, its return value will replace the method call in the expression. If the keyword *void* precedes the method name, the method does not return a value. Because methods with a *void* return type have no value, they cannot be used in an expression; instead, a method call to a method with a *void* return type is a complete statement.

Another keyword we will see preceding the method call in an API is *public*. This keyword means that any client of the class can call this method. If the keyword *private* precedes the method name, only other methods of that class can call that method. Although we will not formally include the *public* keyword in the API, all the methods we discuss in this chapter are *public*.

To call a method for an object of a class, we use **dot notation**, as follows:

```
objectReference.methodName( argument list )
or
objectReference.methodName( arg1, arg2, arg3, ... )
```

Description

COMMON ERROR TRAP

When calling a method that takes no arguments, remember to include the empty parentheses after the method's name. The parentheses are required even if there are no arguments.

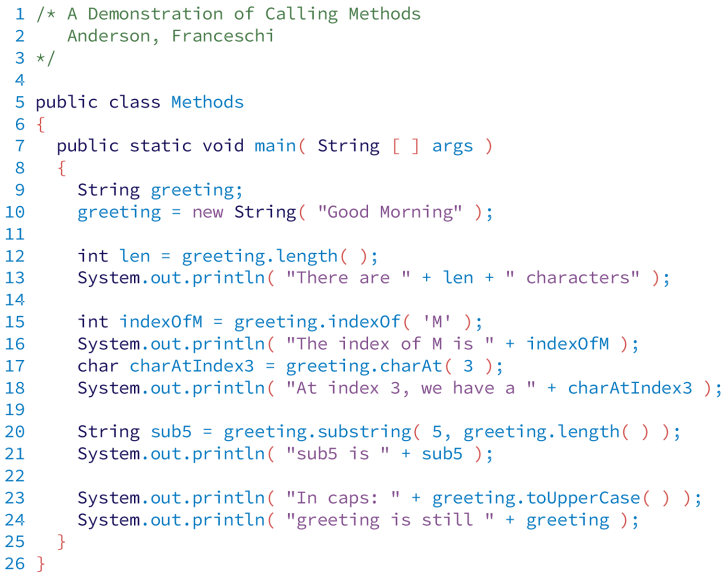
where arg1, arg2, arg3, . . . is the argument list. The object reference is followed immediately by a **dot** (a period), which is followed immediately by the method name. (Later in the chapter, when we call *static* methods, we will substitute the class name for the object reference.) The arguments for the method are enclosed in parentheses.

COMMON ERROR TRAP

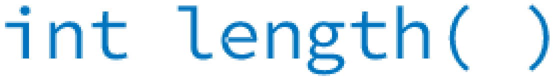
When calling a method, include only values or expressions in your argument list. Including data types in your argument list will cause a compiler error.

Example 3.2 illustrates how to use some methods of the *String* class. At line 12, we call the *length* method with the *greeting* object reference. The API of the *length* method is:

Example 3.2 Calling Methods

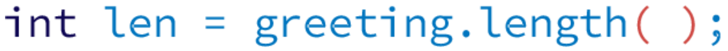


```
 1 /* A Demonstration of Calling Methods
 2    Anderson, Franceschi
 3 */
 4
 5 public class Methods
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     String greeting;
10     greeting = new String( "Good Morning" );
11
12     int len = greeting.length( );
13     System.out.println( "There are " + len + " characters" );
14
15     int indexOfM = greeting.indexOf( 'M' );
16     System.out.println( "The index of M is " + indexOfM );
17     char charAtIndex3 = greeting.charAt( 3 );
18     System.out.println( "At index 3, we have a " + charAtIndex3 );
19
20     String sub5 = greeting.substring( 5, greeting.length( ) );
21     System.out.println( "sub5 is " + sub5 );
22
23     System.out.println( "In caps: " + greeting.toUpperCase( ) );
24     System.out.println( "greeting is still " + greeting );
25   }
26 }
```



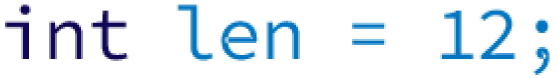
```
int length( )
```

The *length* accepts no parameter; thus, we call it with an empty argument list. It returns an *int*; thus, at line 12, we can assign the value that it returns to an *int* variable, as in:



```
int len = greeting.length( );
```

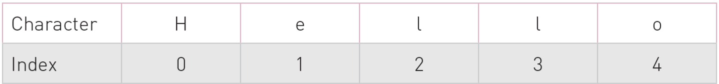
When line 12 is executed, control transfers to the *length* method. When the *length* method finishes executing, the value it returns (*12* in this example) replaces the method call in the statement. The statement effectively becomes:



```
int len = 12;
```

Before discussing the remaining methods in Table 3.2, we need to introduce the concept of **indexes**. Each character in a *String* is assigned an index representing its position within the *String*. The index of the first character is always 0. The index of the second character is always 1, and so on. Therefore, it stands to reason that the index of the last character is always the length of the *String* – 1. The following chart is the *String*

Hello and the indexes of its characters. As we can see, the index of the first letter, *H*, is 0, and the index of the last letter, *o*, is 4, which is one less than the length of the *String* (5).

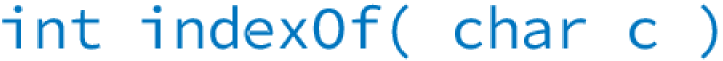


Character	H	e	l	l	o
Index	0	1	2	3	4

Description

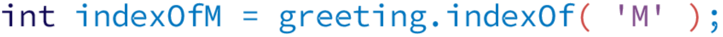
Several of the *String* methods use these indexes to identify which characters within the *String* to process.

At line 15, we call the *indexOf* method (as discussed later in the chapter, there can be several methods with the same name) with the *greeting* object reference. The API of the *indexOf* method is:



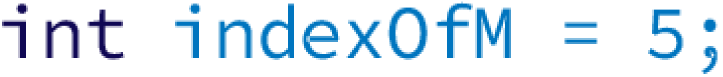
```
int indexOf( char c )
```

The *indexOf* accepts one parameter, a *char*; thus, we must call it with exactly one argument, of type *char*. In this example, we pass the value 'M'. It returns an *int*; thus, at line 15, we can assign the value that it returns to an *int* variable, as in:



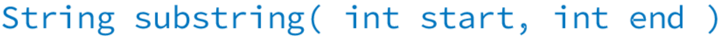
```
int indexOfM = greeting.indexOf( 'M' );
```

When line 15 is executed, control transfers to the *indexOf* method. When the *indexOf* method finishes executing, the value it returns (*5* in this example) replaces the method call in the statement. The statement effectively becomes:



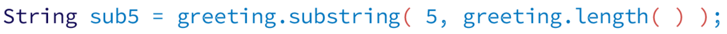
```
int indexOfM = 5;
```

At line 20, we call the *substring* method with the *greeting* object reference. The API of the *substring* method is:



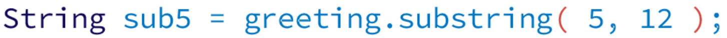
```
String substring( int start, int end )
```

The *substring* method accepts two parameters, both *ints*; thus, we must call it with exactly two arguments, of type *int*. In this example, we pass the values *5* and the *int* value returned by greeting.length( ). The *substring* method returns a *String*; thus, at line 20, we can assign the value that it returns to a *String* variable, as in:



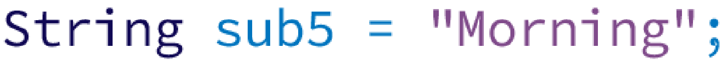
```
String sub5 = greeting.substring( 5, greeting.length( ) );
```

When line 20 is executed, first control transfers to the *length* method. When the *length* method finishes executing, the value it returns (*12* in this example) replaces the method call in the statement. The statement first becomes:



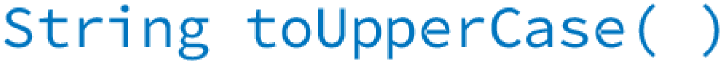
```
String sub5 = greeting.substring( 5, 12 );
```

Control then transfers to the *substring* method, passing the arguments *5* and *12*. When the *substring* method finishes executing, the value it returns (*Morning* in this example) replaces the method call in the statement. The statement becomes:



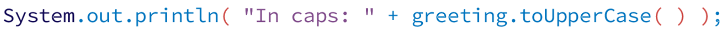
```
String sub5 = "Morning";
```

At line 23, we call the *toUpperCase* method with the *greeting* object reference.



```
String toUpperCase( )
```

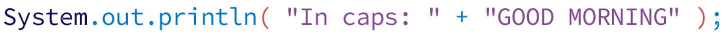
The *toUpperCase* accepts no parameter; thus, we call it with an empty argument list. It returns a *String*; thus, at line 23, we can use the *String* value that it returns in an expression, as in:



```
System.out.println( "In caps: " + greeting.toUpperCase( ) );
```

Description

When line 23 is executed, control transfers to the *toUpperCase* method. When the *toUpperCase* method finishes executing, the value it returns (“GOOD MORNING” in this example) replaces the method call in the statement. The statement effectively becomes:



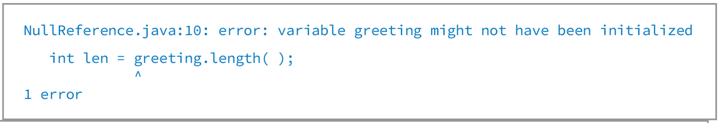
```
System.out.println( "In caps: " + "GOOD MORNING" );
```

Description

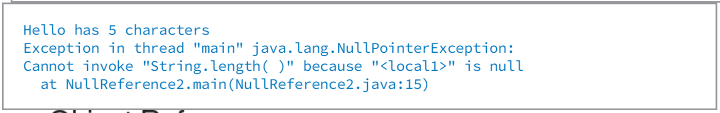
Note that the call to *toUpperCase* returns a copy of *greeting* in uppercase but does not modify the value of *greeting*. **Figure 3.3** shows the output of Example 3.2.

Figure 3.3 Output of Example 3.2

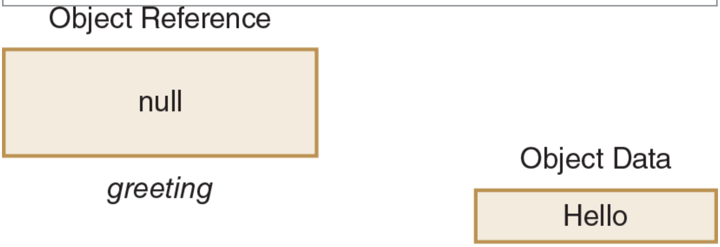
Description



```
NullReference.java:10: error: variable greeting might not have been initialized
   int len = greeting.length( );
             ^
1 error
```



```
Hello has 5 characters
Exception in thread "main" java.lang.NullPointerException:
Cannot invoke "String.length( )" because "<local1>" is null
  at NullReference2.main(NullReference2.java:15)
```



3.4 Using Object References

As we have mentioned, an object reference points to the data of an object. The object reference and the object data are distinct entities. An object can have more than one object reference pointing to it, or an object can have no object references pointing to it.

When an object no longer has an object reference pointing to it, it is marked for **garbage collection**. The **garbage collector**, which is part of the JVM, releases the memory allocated to objects that no longer have an object reference pointing to them.

When an object reference is first declared, but has not yet been assigned to an object, its value is a special literal value, **null**.

If we attempt to call a method using an object reference whose value is *null*, Java generates either a compiler error or a run-time error called an **exception**. The exception is a *NullPointerException* and results in a series of messages printed on the Java console indicating where in the program the *null* object reference was used. Line 10 of **Example 3.3** will generate a compiler error, as shown in **Figure 3.4**, because *greeting* has not been instantiated.

Example 3.3 Attempting to Use a *null* Object

Reference

```java
/* A Demonstration of Trying to Use a Null Object Reference
   Anderson, Franceschi
*/

public class NullReference
{
  public static void main( String [ ] args )
  {
    String greeting;
    int len = greeting.length( );
  }
}
```

Description

Figure 3.4 Compiler Error from Example 3.3

Description

Java does not provide support for explicitly deleting an object. One way to indicate to the garbage collector that our program is finished with an object is to set its object reference to *null*. Obviously, once an object reference has the value *null*, it can no longer be used to call methods.

COMMON ERROR TRAP

Using a *null* object reference to call a method will generate either a compiler error or a *NullPointerException* at run time. Be sure to instantiate an object before attempting to use the object reference.

Example 3.4 shows a *NullPointerException* being generated at run time. Line 9 instantiates the *greeting* object, and lines 10–11 print its number of characters. Line 13 assigns *null* to the object reference, and line 15 attempts to call the *length* method again. As **Figure 3.5** shows, a *NullPointerException* is generated. Notice that the console message indicates the name of the application class (*NullReference2*), the method *main*, and the line number *15*, where the exception occurred. The JVM often prints additional lines in the message, depending on where in our program the error occurred.

Example 3.4 Another Attempt to Use a *null* Object Reference

```
/* A Demonstration of Trying to Use a Null Object Reference
   Anderson, Franceschi
*/

public class NullReference2
{
  public static void main( String [ ] args )
  {
    String greeting = new String( "Hello" );
    int len = greeting.length( );
    System.out.println( greeting + " has " + len + " characters" );

    greeting = null;
    // attempt to use null object reference
    len = greeting.length( );
  }
}
```

Description

Figure 3.5 Output of Example 3.4

Description

Figure 3.6 shows the *greeting* object reference and object data after setting the object reference to *null*.

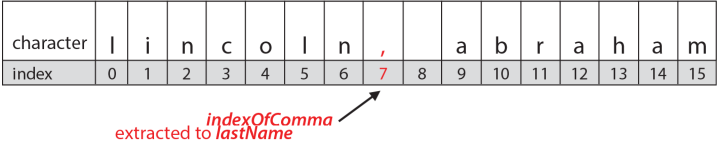
Figure 3.6 The *greeting* Object Reference Set to *null*

Description

```
s1 is: OOP in Java
s2 is: is not that difficult.
s1 + s2 is: OOP in Java is not that difficult.
s1 is still: OOP in Java
s2 is still: is not that difficult.

The length of the empty String is 0

The first and last characters of Hello are H and o

The index of e is 1
The index of l is 2
The index of lo is 3
The index of h is -1

The middle three characters of Hello are ell
All characters of Hello except the first are ello

Is s4 blank? true
Is s4 now empty? true
```

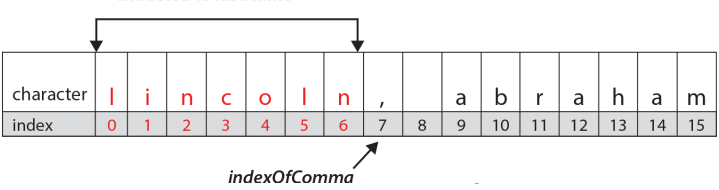
```
The index of ',' in "lincoln, abraham" is 7
The last name is lincoln
The first name is abraham

The formatted full name is Abraham Lincoln
```

character	l	i	n	c	o	l	n	,		a	b	r	a	h	a	m
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



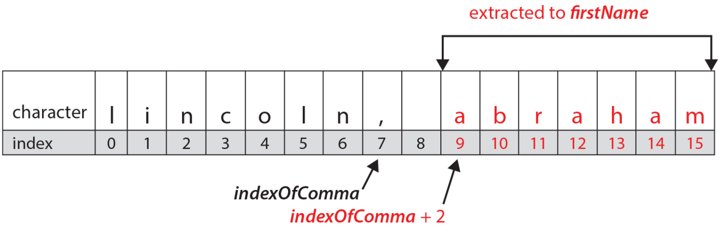
character	l	i	n	c	o	l	n	,		a	b	r	a	h	a	m
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

indexOfComma



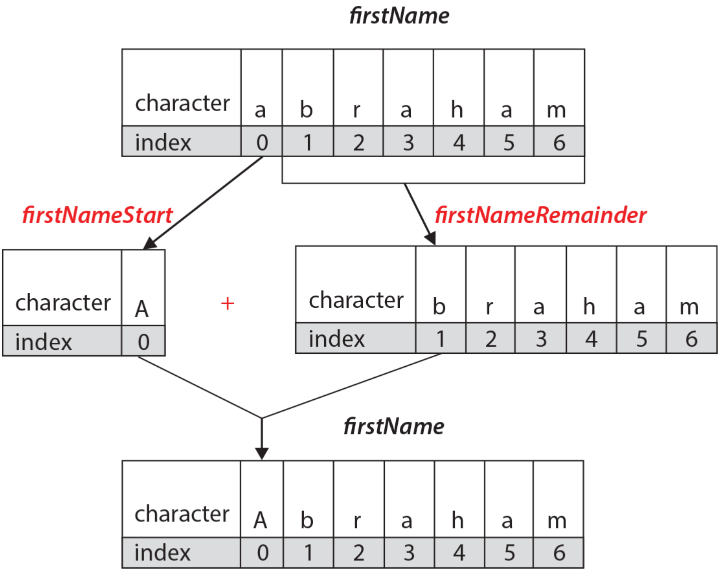
extracted to *lastName*
indexOfComma

character	l	i	n	c	o	l	n	,		a	b	r	a	h	a	m
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



extracted to *firstName*
indexOfComma
indexOfComma + 2

character	l	i	n	c	o	l	n	,		a	b	r	a	h	a	m
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15


firstName
character
a
b
r
a
h
a
m
index
0
1
2
3
4
5
6
firstNameStart
firstNameRemainder
character
A
index
0
+
character
b
r
a
h
a
m
index
1
2
3
4
5
6
firstName
character
A
b
r
a
h
a
m
index
0
1
2
3
4
5
6

3.5 More on the *String* Class

As we've discussed, Java provides the *char* primitive data type, which stores one character. Almost every program, however, needs a data type that stores more than one character. Programs need to process names, addresses, or labels of many kinds. For example, many programs involve a login procedure where the user has to enter a user ID and a password. The program reads the user ID and password, compares them to values stored in a database, and allows the user to continue only if the user ID and password match the database values.

To handle this type of data, Java provides a *String* class. Because the *String* class is part of the *java.lang* package, it is automatically available to any Java program and we do not need to use the *import* statement. As explained earlier, the *String* class provides several constructors, as well as a number of methods to manipulate, search, and compare *String* objects. In addition, the Java language provides two concatenation operators to combine *String* objects with variables, literals, or other *String* objects.

Example 3.5 shows how to use these tools in a program.

Example 3.5 Demonstrating *String* Methods

```
/* Demonstrating String methods
   Anderson, Franceschi
*/
public class StringDemo
{
  public static void main( String [ ] args )
  {
    String s1 = new String( "OOP in Java " );
    System.out.println( "s1 is: " + s1 );
    String s2 = "is not that difficult. ";
    System.out.println( "s2 is: " + s2 );

    String s3 = s1 + s2; // new String is s1, followed by s2
    System.out.println( "s1 + s2 is: " + s3 );

    System.out.println( "s1 is still: " + s1 ); // s1 is unchanged
    System.out.println( "s2 is still: " + s2 ); // s2 is unchanged

    String empty = new String( );
    System.out.println( "\nThe length of the empty String is "
                        + empty.length( ) );

    String greeting = "Hello"; // instantiate greeting
    char firstChar = greeting.charAt( 0 );
    char lastChar = greeting.charAt( greeting.length( ) - 1 );
    System.out.println( "\nThe first and last characters of "
                        + greeting + " are " + firstChar
                        + " and " + lastChar );

    int indexOfE = greeting.indexOf( 'e' );
    System.out.println( "\nThe index of e is " + indexOfE );
    System.out.println( "The index of l is "
                           + greeting.indexOf( 'l' ) );
    System.out.println( "The index of lo is "
                           + greeting.indexOf( "lo" ) );
    System.out.println( "The index of h is "
                           + greeting.indexOf( 'h' ) );

    System.out.println( "\nThe middle three characters of Hello are "
                           + greeting.substring( 1, 4 ) );
    System.out.println( "All characters of Hello except the first are "
                           + greeting.substring( 1 ) );

    String s4 = "\n\t   \t    \n \t  ";
    System.out.println( "\nIs s4 blank? " + s4.isBlank( ) );
    s4 = s4.strip( );
    System.out.println( "Is s4 now empty? " + s4.isEmpty( ) );
  }
}
```

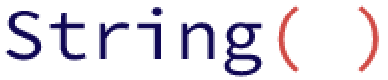
Description

When this program runs, it will produce the output shown in **Figure 3.7**.

Figure 3.7 Output from Example 3.5

Description

The default constructor



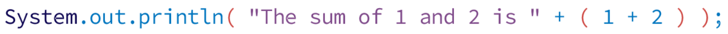
```
String( )
```

creates an empty *String*—in other words, a *String* containing no characters. We can add characters to the *String* later. This constructor will come in handy in programs where we build up our output, piece by piece. Line 19 uses the second constructor to instantiate an empty *String* named *empty*. Do not confuse a *String* that has not been instantiated, and therefore is *null*, with a *String* that has been instantiated and is empty. At line 21, we call the *length* method with *empty*. It returns the value *0*, since *empty* has no character.

Additionally, because *Strings* are used so frequently in programs, Java provides special support for instantiating *String* objects without explicitly using the *new* operator. We can simply assign a *String* literal to a *String* object reference. For example, at line 10, we assign a *String* literal to the *s2 String* reference.

Java also provides special support for appending a *String* to the end of another *String* through the **concatenation operator (+)** and the **shortcut version of the concatenation operator (+=)**. This concept is illustrated in Example 3.5. Lines 8–11 declare, instantiate, and print two *String* objects, *s1* and *s2*. Line 13 concatenates *s1* and *s2*, and the resulting *String* is assigned to the *s3 String* reference, which is printed at line 14. Finally, we output *s1* and *s2* again at lines 16 and 17 to illustrate that their values have not changed.

Note that the *String* concatenation operator is the same character as the addition arithmetic operator. In some cases, we need to make clear to the compiler which operator we want to use. For example, this statement uses both the *String* concatenation operator and the addition arithmetic operator:



```
System.out.println( "The sum of 1 and 2 is " + ( 1 + 2 ) );
```

Description

Notice that we put *1 + 2* inside parentheses to let the compiler know that we want to add two *ints* using the addition arithmetic operator (+). The addition will be performed first because of the higher operator precedence of parentheses.

Then it will become clear to the compiler that the other + operator is intended to be a *String* concatenation operator because its operands are a *String* and an *int*.

Example 3.5 also demonstrates some useful methods of the *String* class, which are summarized in Table 3.2.

The *length* Method

The *length* method returns the number of characters in a *String*, including any spaces and punctuation. Sometimes, the number of characters in a user ID is limited, for example, to eight, and this method is useful to ensure that the length of the ID does not exceed the limit.

The *length* method is called using a *String* object reference and the dot operator. As we can see in Table 3.2, the *length* method does not take any arguments and returns an *int*, which contains the number of characters in the *String*. We can output the return value or we can store the return value in an *int* variable. In Example 3.5, at line 21, we call the *length* method inside the output statement. The return value replaces the method call, so the result is that we output the number of characters—in this case 0, as seen in Figure 3.7.

The *charAt* Method

The *charAt* method returns the character at a specified index in a *String*.

One of the uses of this method is for extracting just the first character of a *String*, which might be advantageous when prompting the user for an answer to a question.

For example, we might ask users if they want to play a game again. They can answer “y,” “yes,” or “you bet!” Our only concern is whether the first character is a *y*, so we could use this method to store just the first character of their answer into a *char* variable. Assuming the user’s answer was previously read and stored into a *String* variable named *answerString*, we could use the following statement to extract the first character of *answerString*:

```
char answerChar = answerString.charAt( 0 );
```

Later in this text, when we talk about selection, we will be able to test whether *answerChar* is a *'y'.*

As shown in Table 3.2, the *charAt* method takes one argument, an *int*, which is the index of the character within the *String* to extract. The method returns that character as a *char*. In Example 3.5, at line 24, we call the *charAt* method with an argument of 0 to extract the first character in *greeting* and store the return value in the *char* variable, *firstChar*. At line 25, we extract the last character in *greeting* by calling the *charAt* method with the argument *greeting.length( ) – 1* and store the return value in the *char* variable, *lastChar*. Although in this case we could have sent an argument of *4* and gotten the same result, sending an argument of *length( ) -1* to the *charAt* method will extract the last character of any *String*. We then output both characters (lines 26–28). Note that we assume that *greeting* has at least one character; otherwise, the expression *greeting.length( ) –1* is equal to *-1*, an invalid index.

The *indexOf* Methods

The same method name can be used for several methods, provided that these methods have a different parameter list. The *indexOf* methods are useful for searching a *String* for a specific character or *String*. Table 3.2 shows two versions of this method, one that accepts a single *char* as the argument and one that accepts a *String* as an argument. In either case, the methods return as an *int* the index of the first occurrence of the *char* or the first occurrence of the first character of the *String* if the *String* is found. If the character or *String* is not found, the *indexOf* methods return –1.

In Example 3.5, we use the *indexOf* method to search the *String greeting* for various characters. First, at line 30, we search *greeting* for the character *'e'* and store the returned index in *indexOfE*. When we output that index at line 31, we see in Figure 3.7 that *'e'* was found at index 1. At lines 32–33, we search *greeting* for the letter *'l'* and output the returned index. Notice that *greeting* has two letter *l*s, but the *indexOf* method returns only the index of the first occurrence, which is 2. Then at lines 34–35, we search *greeting* for the *String "lo"*, which the *indexOf* method finds beginning at index 3. Finally, at lines 36–

37, we search *greeting* for the lowercase letter '*h*', which is not in our *String*. We see when outputting the return value that the method returned −1, indicating that the *indexOf* method did not find a lowercase *h* in *greeting*.

The *substring* Methods

The *substring* methods are useful for creating a new *String* from characters in an existing *String*. The *substring* method returns a group of characters, or **substring**, from a *String*. The original *String* is unchanged. Table 3.2 shows two versions of the *substring* method. The first version takes two *int* arguments. The first argument is the index at which to start extracting the characters, and the second argument is the index of the first character not to extract. Thus, the *endIndex* argument is one position past the last character to extract. We know this sounds a little awkward, but setting up the arguments this way actually makes the method easier to use, as we will demonstrate.

In Example 3.5, we create and output a new *String* containing the middle three characters of *greeting* (lines 39–40). The first argument specifies that the new *String* should start with the character at index 1, which is an '*e*', and the *String* should include all succeeding characters up to but not including the character at index 4. The result is a *String* containing "*ell*".

The second version of the *substring* method is convenient for extracting all characters in a *String* from one position to the end of the *String*. This version of *substring* takes only one *int* argument, which is the index at which to start collecting characters. The returned *String* will consist of all characters starting at *startIndex* and continuing to the end of the *String*. We call this version at lines 41–42, where we extract and output all characters in *greeting* except the first.

The *isBlank* and *strip* Methods

The *isBlank* method is useful to check if a *String* does not contain any character other than white space characters (including tabs and newline characters). We can use the *strip* method to delete all the leading and trailing white space characters. We demonstrate them at lines 44–47.

String Processing

With the popularity of large search engines and Big Data, the ability to find, extract, and reformat information in a *String* of characters is becoming an important part of computer science. We call this **string processing**. For example, using the *String* methods we have just discussed, we can write an application to extract and format a first and last name from a *String*. Assume we know that a *String* contains a name in this format:

```
<lastname>, <firstname>
```

that is, with the last name first, followed by a comma and a space, then the first name.

Our job is to produce a *String* in this format:

```
<Firstname> <Lastname>
```

with a space separating the first and last names and with the first character of each name capitalized and all other letters in lowercase.

Example 3.6 shows the code to accomplish this, and its output is shown in **Figure 3.8**. In this example, we have hard-coded the name in the *String invertedName* (line 8) to be: “lincoln, abraham” but the code we write needs to work for any name in this format.

Example 3.6 ***StringProcessing.java***

```java
/* Demonstrating String Processing
   Anderson, Franceschi
*/
public class StringProcessing
{
  public static void main( String [ ] args )
  {
    String invertedName = "lincoln, abraham";

    int comma = invertedName.indexOf( ',' ); // find the comma
    System.out.println( "\nThe index of ',' in \""
                        + invertedName + "\" is " + comma );

    // last name: extract all characters before the comma
    String lastName = invertedName.substring( 0, comma );
    System.out.println( "The last name is " + lastName );

    // first name: extract all characters after the space
    String firstName = invertedName.substring( comma + 2 );
    System.out.println( "The first name is " + firstName );

    // convert the first letter of each name to uppercase
    String firstNameStart = firstName.substring( 0, 1 );
    firstNameStart = firstNameStart.toUpperCase( );
    String firstNameRemainder = firstName.substring( 1 );
    firstNameRemainder = firstNameRemainder.toLowerCase( );
    firstName = firstNameStart + firstNameRemainder;

    String lastNameStart = lastName.substring( 0, 1 );
    lastNameStart = lastNameStart.toUpperCase( );
    String lastNameRemainder = lastName.substring( 1 );
    lastNameRemainder = lastNameRemainder.toLowerCase( );
    lastName = lastNameStart + lastNameRemainder;

    String fullName = firstName + " " + lastName;
    System.out.println( "\nThe formatted full name is "
                           + fullName );
  }
}
```

Description

Figure 3.8 The Output of Example 3.6

Description

REFERENCE POINT

You can read more about the *String* class on Oracle's Java website *www.oracle.com/technetwork/java*.

Let's look at the characters and indexes of this *String*. (Again, this is just one example. We are writing code to work with any name in this format.)

Description

Let's begin by extracting the last name. We know the last name starts at the beginning of the *String* and ends at the character before the comma, so we need to find the comma. As we have seen, the *indexOf* method allows us to search for the location of a character. At line 10, we call the *indexOf* method, sending it a comma as the search argument, and storing the return value in the *int* variable, *indexOfComma*. We output the *indexOfComma* at lines 11–12 to verify that we have indeed found the comma at index 7.

Description

We then use the *substring* method to extract all the characters from the first character up to, **but not including**, the location of the comma. Our two arguments, then, are 0 as the start index, and *indexOfComma* as the end index. At line 16, we output the extracted last name.

Description

Next, we want to extract the first name. Looking at our inverted name above, we see that the first name starts at the character after the space (index 9 in this case) and continues to the end of the *String*. We could either search for the space, or we could specify the starting index for the first name as an offset from the location of the comma that we have already found. Let's do the latter.

Because we want to extract all characters starting two positions after the comma until the end of the *String*, we can use the version of the *substring* method that accepts one argument. At line 19, we call the *substring* method with the argument *indexOfComma + 2*. This will extract all characters from index 9 to the end of *invertedName*. We output the returned *String*, *firstName*, at line 20.

Description

We have now isolated the first and last names in their own

Strings.

We have one more task to perform: formatting the capitalization of the names. As mentioned, we want to capitalize the first character of each name and make the rest of the characters lowercase. We can do this by dividing each name into two *Strings*: one *String* that will contain just the first character, and the second *String* that will contain the remaining characters. At line 23, we extract just the first character of the first name using the *substring* method with the start index being 0 and the end index being 1. We then capitalize the first letter by calling the *toUpperCase* method on that *String* and storing the return value back into *firstNameStart* (line 24). We again call the *substring* method to extract the remaining characters of the first name (line 25), starting at index 1 and continuing to the end of the *String* and convert that *String* to lowercase by calling the *toLowerCase* method *(line 26)*, storing the result back into the *firstNameRemainder String*. We can now combine those two *Strings* into a formatted first name using the *String* concatenation operator (line 27).

At lines 29–33, we perform the corresponding operations on the last name.

COMMON ERROR TRAP

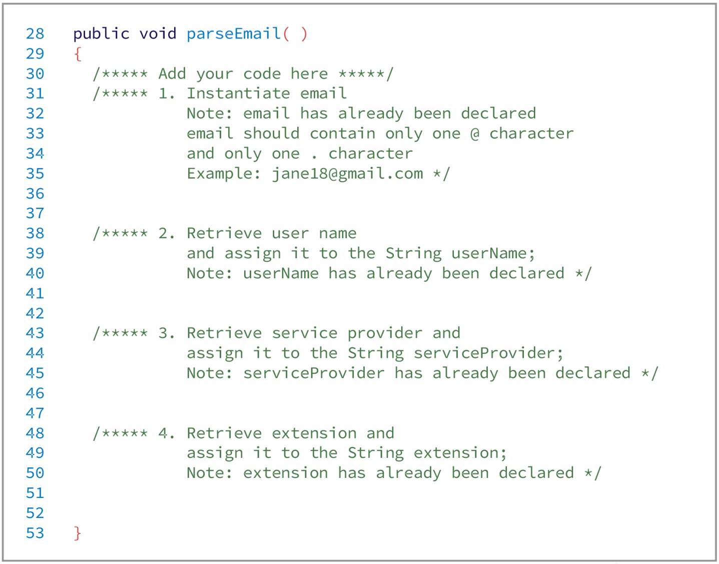
Specifying a negative start index or a start index past the last character of the *String* will generate a *StringIndexOutOfBoundsException*. Specifying a negative end index or an end index greater than the length of the *String* will also generate a *StringIndexOutOfBoundsException*.

At this point, the formatting of the name is complete and we concatenate the first name, a space, and the last name to form the full name (line 35), which we output at lines 36–37.

Description

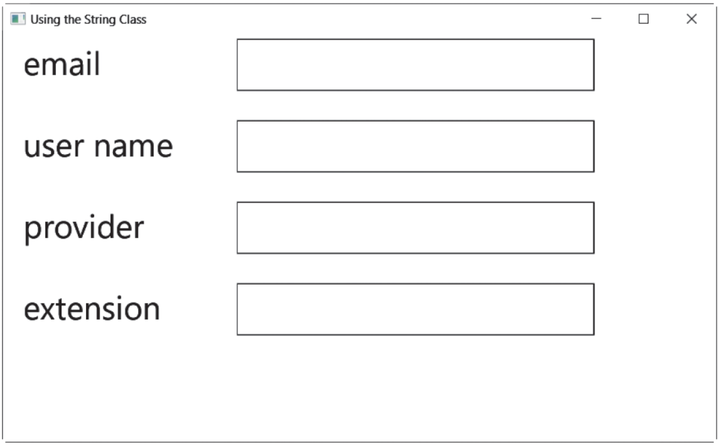
Try running this example with your name and with a mixture of upper- and lowercase letters to verify that it does indeed work with other names.

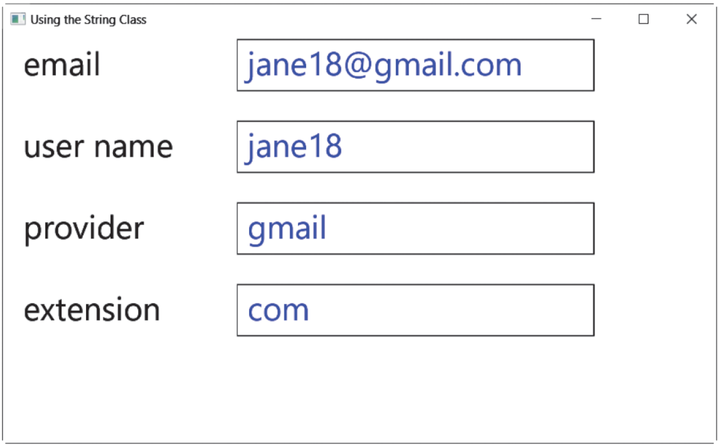
When we are calculating indexes and the number of characters to extract, we need to be careful not to specify an index that is not in the *String*, because that will generate a run-time error, *StringIndexOutOfBoundsException*.



```
public void parseEmail( )
{
  /***** Add your code here *****/
  /***** 1. Instantiate email
            Note: email has already been declared
            email should contain only one @ character
            and only one . character
            Example: jane18@gmail.com */


  /***** 2. Retrieve user name
            and assign it to the String userName;
            Note: userName has already been declared */


  /***** 3. Retrieve service provider and
            assign it to the String serviceProvider;
            Note: serviceProvider has already been declared */


  /***** 4. Retrieve extension and
            assign it to the String extension;
            Note: extension has already been declared */


}
```

```
private String email; // String object reference
```





email jane18@gmail.com

user name jane18

provider gmail

extension com

3.6 Programming Activity 1: Calling Methods

In this Programming Activity, you'll use a program that parses a *String* storing an email and retrieves the user name, service provider, and extension. You should assume that there is exactly one @ character and one dot (.) character in the email, and that the @character is before the dot (.) character. For example, if the email is *Jane18@gmail.com*, you need to retrieve *Jane18*, *gmail*, and *com*. If the email is *Bob21@government.uk*, you need to retrieve *Bob21*, *government*, and *uk*.

Copy all the files in this chapter's Programming Activity 1 folder in the supplied code files to a folder on your computer. Note that all files should be in the same folder.

Open the *StringDrawing.java* source file. You'll notice that the class already contains some source code. Your job is to fill in the blanks. Search for five asterisks in a row (*****). This will position you to the places in the source code where you will add your code. This section of code is shown in **Figure 3.9**.

Figure 3.9 Partial Listing of *StringDrawing.java*

Description

Notice that line 25 in the source file is a declaration of a *String* object reference, *email*.

You will use this object reference for instantiating an object and for calling the methods of the *String* class.

In the source file, you will see four commented lines that instruct you to instantiate the object or call a method.

To complete the Programming Activity, write the requested code on the lines after the numbered instructions.

Compile *StringDrawing.java* and run *StringApplication* and you will see a window that looks like the one in **Figure 3.10**.

Figure 3.10 Programming Activity 1 Output

Description

Write the code for the remaining instructions, compiling and running the program after completing each task. The program will display your results, as shown in **Figure 3.11**.

Figure 3.11 Finished Programming Activity 1 Output

Description

DISCUSSION QUESTIONS

1. After instructions 6 and 7 have executed, why are the day and month values not changed?
2. How would your code change if we allowed one or more dot (.) characters before the @character in the email (in addition to the dot character after the @character)?

3.7 The Java Class Library

Java provides more than 2,000 predefined classes that we can use to add functionality to our program. In this chapter, in addition to the *String* class, we'll discuss a few commonly used Java classes:

- *Random*, which generates random numbers
- *Scanner*, which provides methods for reading input from the keyboard
- *System* and *PrintStream*, which provide data members and methods for printing data on the Java console
- *DecimalFormat* and *NumberFormat*, which allow us to format numbers for output
- *Math*, which provides methods for performing mathematical operations
- Object wrappers, which provide an object equivalent to primitive data types so they can be used in our program as if they were objects

The Java classes are arranged in **packages**, grouped according to functionality.

Table 3.3 describes some of the Java packages that we will cover in this text. We can find more details on these classes on Oracle's Java website: www.oracle.com/technetwork/java.

Commonly Used Java Packages

Package	**Categories of Classes**
java.lang	Basic functionality common to many programs, such as the *String* class, *Math* class, and object wrappers for the primitive data types

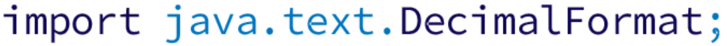
java.text	Classes for formatting numeric output
java.util	The *Scanner* class, the *Random* class, and other miscellaneous classes
java.io	Classes for reading from and writing to files

Description

Many of the commonly used classes, such as *String* and *Math*, reside in the *java.lang* package. Any class in the *java.lang* package is automatically available to our program.

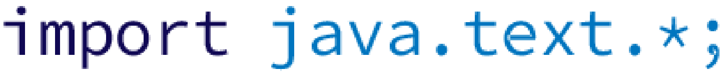
To use a class from the Java Class Library that is not in the *java.lang* package, we need to tell the compiler in which package the class resides; in other words, we need to tell the compiler where to find the class. To do this, we include an **import** statement in our program. The *import* statement is inserted at the top of the program after our introductory comments, but before the *class* statement that begins the program.

For example, if we want to use the *DecimalFormat* class to format a floating-point number for output, we would import the *DecimalFormat* class from the *java.text* package as follows:



```
import java.text.DecimalFormat;
```

If we're using more than one class from a package, we can import all those classes we use by replacing the class name with an asterisk, as in the following example:



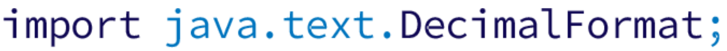
```
import java.text.*;
```

```
The first price is: $78.67
The second price is: $34.50
The third price is: $0.33
The fourth price is: $3.00
The fifth price is: $100.23
The first average is: 98.0%
The second average is: 98.75%
The time is 05:12:00
millions is 1,234,567
```

3.8 Formatting Output with the *DecimalFormat* Class

In a computer program, numbers represent a real-life entity, for instance, a price or a winning percentage. Floating-point numbers, however, are calculated to many decimal places and, as a result of some computations, can end up with more significant digits than our programs need. For example, the price of an item after a discount could look like 3.46666666666666, when all we really want to display is $3.47; that is, a leading dollar sign and two significant digits after the decimal point. The *DecimalFormat* class allows us to specify the number of digits to display after the decimal point and to add dollar signs, commas, and percentage signs (%) to our output.

The *DecimalFormat* class is part of the *java.text* package, so to use the *DecimalFormat* class, we include the following *import* statement in our program:



```
import java.text.DecimalFormat;
```

Description

We can instantiate a *DecimalFormat* object using a simple constructor that takes a *String* object as an argument. This *String* object represents how we want our formatted number to look when it's printed. The API for that constructor is shown in **Table 3.4**.

A *DecimalFormat* Constructor and the *format* Method

DecimalFormat Class Constructor

`DecimalFormat( String pattern )`

instantiates a *DecimalFormat* object with the output pattern specified in the argument

The *format* Method

Return value	**Method name and argument list**
`String`	`format( double number )` returns a *String* representation of *number* formatted according to the *DecimalFormat* object used to call the method

Description

The pattern that we use to instantiate the *DecimalFormat* object consists of special characters and symbols and creates a "picture" of how we want the number to look when printed. Some of the more commonly used symbols and their meanings are listed in **Table 3.5**.

Special Characters for *DecimalFormat* Patterns

Common Pattern Symbols for a *DecimalFormat* object

Symbol	**Meaning**
0	Required digit. Do not suppress leading or terminating 0s in this position.
#	Optional digit. Do not include a leading or terminating digit that is 0.

.	Decimal point.
,	Comma separator.
$	Dollar sign.
%	Multiply by 100 and append a percentage sign.

Description

Once we have instantiated a *DecimalFormat* object, we format a number by passing it as an argument to the *format* method, shown in Table 3.4. **Example 3.7** demonstrates the use of the *DecimalFormat* patterns and calling the *format* method. The output for this program is shown in **Figure 3.12**.

Example 3.7 Demonstrating the *DecimalFormat* Class

```java
/* Demonstrating the DecimalFormat class
   Anderson, Franceschi
*/

// import the DecimalFormat class from the java.text package
import java.text.DecimalFormat;

public class DemoDecimalFormat
{
  public static void main( String [ ] args )
  {
    // first, instantiate a DecimalFormat object specifying a
    // pattern for currency
    DecimalFormat pricePattern = new DecimalFormat( "$0.00" );

    double price1 = 78.66666666;
    double price2 = 34.5;
    double price3 = .3333333;
    int price4 = 3;
    double price5 = 100.23;

    // then print the values using the pattern
    System.out.println( "The first price is: "
                  + pricePattern.format( price1 ) );
    System.out.println( "\nThe second price is: "
                  + pricePattern.format( price2 ) );
    System.out.println( "\nThe third price is: "
                  + pricePattern.format( price3 ) );
    System.out.println( "\nThe fourth price is: "
                  + pricePattern.format( price4 ) );
    System.out.println( "\nThe fifth price is: "
                  + pricePattern.format( price5 ) );

    // instantiate another new DecimalFormat object
    // for printing percentages
    DecimalFormat percentPattern = new DecimalFormat( "0.0#%" );

    double average1 = .98;
    System.out.println( "\nThe first average is: "
                  + percentPattern.format( average1 ) );
    double average2 = .98748;
    System.out.println( "\nThe second average is: "
                  + percentPattern.format( average2 ) );

    // now instantiate another new DecimalFormat object
    // for printing time as two digits
    DecimalFormat timePattern = new DecimalFormat( "00" );

    int hours = 5, minutes = 12, seconds = 0;
    System.out.println( "\nThe time is "
                  + timePattern.format( hours ) + ":"
                  + timePattern.format( minutes ) + ":"
                  + timePattern.format( seconds ) );

    // now instantiate another DecimalFormat object
    // for printing numbers in the millions.
    DecimalFormat bigNumber = new DecimalFormat( "#,###" );

    int millions = 1234567;
    System.out.println( "\nmillions is "
                  + bigNumber.format( millions ) );
  }
}
```

Description

Figure 3.12 Output from Example 3.7

Description

In Example 3.7, line 14 instantiates the *DecimalFormat* object, *pricePattern*, which will be used to print prices. In the pattern

```
"$0.00"
```

Description

the first character of this pattern is the dollar sign ($), which we want to precede the price. The 0 specifies that there should be at least one digit to the left of the decimal point. If there is no value to the left of the decimal point, then insert a zero. The two 0s that follow the decimal point specify that exactly two digits should be to the right of the decimal point; that is, if more than two digits are to the right of the decimal point, round to two digits; if the last digit is a 0, include the zero, and if there is no fractional part to the number, include two zeroes. Using this pattern, we see that in lines 23–24, *price1* is printed to two decimal places. In lines 25–26, *price2* is printed with a zero in the second decimal place.

In lines 29–30, we print *price4*, which is an integer. The *format* method API calls for a *double* as the argument; however, because all numeric data types can be promoted to a *double*, any numeric data type can be sent as an argument. The result is that two zeroes are added to the right of the decimal point.

COMMON ERROR TRAP

When using a % symbol to *format* a percentage, do not multiply the value by 100. The *format* method will do that for you.

Finally, we use the *pricePattern* pattern to print *price5* in lines 31–32, which needs no rounding or padding of extra digits.

Next, line 36 instantiates a *DecimalFormat* object, *percentPattern*, for formatting percentages to one or two decimal places (`"0.0#%"`). Lines 38–40 and 41–43 define the variables *average1* and *average2* then print them using the *format* method in an output statement. Notice that we do not multiply the averages by 100 before we format the percentage. A % in the pattern signals to the *format* method that it should multiply the value by 100 before converting the number to a *String*.

REFERENCE POINT

You can read more about the *DecimalFormat* class on Oracle's Java website: *www.oracle.com/technetwork/java*.

Line 47 defines another pattern, `"00"`, which is useful for printing the time with colons between the hour, minutes, and seconds. When the time is printed on lines 50–53, the hours, minutes, and seconds are padded with a leading zero, if necessary.

Line 57 defines our last pattern, `"#,###"`, which can be used to insert commas into integer values in the thousands and above. Lines 60–61 print the variable *millions* with commas separating the millions and thousands digits. Notice that the pattern is extrapolated for a number that has more digits than the pattern.

```
The die roll is 2
The random number between 20 and 200 is 117
```

3.9 Generating Random Numbers with the *Random* Class

Random numbers come in handy for many operations in a program, such as rolling dice, dealing cards, timing the appearance of a nemesis in a game, or other simulations of seemingly random events.

There's one problem in using random numbers in programs, however: Computers are **deterministic**. In essence, this means that given a specific input to a specific set of instructions, a computer will always produce the same output. The challenge, then, is generating random numbers while using a deterministic system. Many talented computer scientists have worked on this problem, and some innovative and complex solutions have been proposed.

The *Random* class, which is in the *java.util* package, uses a mathematical formula to generate a sequence of numbers. The constructor generates a **seed** value, which determines where in that sequence the set of *random* numbers will begin. As such, the *Random* class generates numbers that appear to be, but are not truly, random. These numbers are called **pseudorandom** numbers, and they work just fine for our purposes.

Table 3.6 shows a constructor for the *Random* class and a method for retrieving a random integer. The default constructor creates a random number generator using a seed value. Once the random number generator is created, we can ask for a random number by calling the *nextInt* method. Other methods, *nextDouble*, *nextBoolean*, *nextByte*, and *nextLong*, which are not shown in Table 3.6, return a random *double, boolean, byte*, or *long* value, respectively.

A *Random* Class Constructor and the *nextInt*

Method

Random Class Constructor

`Random( )`

creates a random number generator with a seed generated using the system time

The *nextInt* Method

Return value	Method name and argument list
`int`	`nextInt( int number )` returns a random integer ranging from 0 up to, but not including, *number* in uniform distribution

Description

To demonstrate how to use the random number generator, let's take rolling a die as an example. To simulate the roll of a six-sided die, we need to simulate random occurrences of the numbers 1 through 6. If we call the *nextInt* method with an argument of 6, it will return an integer between 0 and 5. To get randomly distributed numbers from 1 to 6, we can simply add 1 to the value returned by the *nextInt* method. Thus, if we have instantiated a *Random* object named *random*, we can generate random numbers from 1 to 6 by calling the *nextInt* method in this way:

```
int die = random.nextInt( 6 ) + 1;
```

In general, assuming than *n* is smaller than *m*, if we want to generate random numbers from *n* to *m*, we should call the *nextInt* method with the number of possible random values (*m* – *n* + 1), and then add the first value of our sequence (*n*) to the returned value. Thus, this statement generates a random number between 10 and 100 inclusive:

```
int randomNumber = random.nextInt( 100 - 10 + 1 ) + 10;
```

Line 18 of **Example 3.8** generates a random number between 20 and 200 inclusive.

Example 3.8 A Demonstration of the *Random* Class

```
/*A demonstration of the Random class
  Anderson, Franceschi
*/
import java.util.Random;

public class RandomNumbers
{
  public static void main( String [ ] args )
  {
    Random random = new Random( );

    // simulate the roll of a die
    int die = random.nextInt( 6 ) + 1;
    System.out.println( "\nThe die roll is " + die );

    // generate a random number between 20 and 200
    int start = 20, end = 200;
    int number = random.nextInt( end - start + 1 ) + start;
    System.out.println( "\nThe random number between " + start
                        + " and " + end + " is " + number );
  }
}
```

Description

REFERENCE POINT

You can read more about the *Random* class on Oracle's Java website: *www.oracle.com/technetwork/java*.

When the *RandomNumbers* program executes, it will produce output similar to **Figure 3.13**. The output will vary from one execution of the program to the next because different random numbers will be generated.

Figure 3.13 Output from Example 3.8

Description

```
Enter your first name > Syed
Your name is Syed

Enter your age as an integer > 21
Your age is 21

Enter your GPA > 3.875
Your GPA is 3.875
```

```
Enter your age as an integer >21
```

```
Enter your middle initial > A
Your middle initial is A
```

```
Enter a sentence > Scanner is useful.
You said: "Scanner is useful."
```

```
Enter your first name > Sarah
Your name is Sarah

Enter your age as an integer > a
Exception in thread "main" java.util.InputMismatchException
   at java.base/java.util.Scanner.throwFor(Scanner.java:939)
   at java.base/java.util.Scanner.next(Scanner.java:1594)
   at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
   at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
   at DataInput.main(DataInput.java:18)
```

3.10 Input from the Console Using the *Scanner* Class

As our programs become more complex, we will need to allow the users of our programs to input data. User input can be read into our program in several ways:

- from the keyboard
- from a file
- through a Graphical User Interface (GUI)

The Java Class Library provides classes for all types of data input. In this chapter, we will concentrate on inputting data from the keyboard.

The *Scanner* class provides methods for reading *byte, short, int, long, float, double*, *boolean*, and *String* data types from the keyboard. These methods are shown in **Table 3.7**.

Selected Methods of the *Scanner* Class

A *Scanner* Class Constructor

`Scanner( InputStream dataSource )`

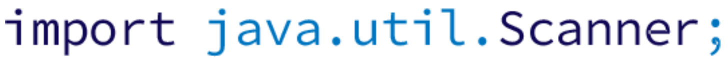
creates a Scanner object that will read from the InputStream dataSource . To read from the keyboard, we will use the predefined InputStream System.in .

Selected Methods of the *Scanner* Class

Return value	Method name and argument list
`byte`	`nextByte( )` returns the next input as a *byte*
`short`	`nextShort( )` returns the next input as a *short*
`int`	`nextInt( )` returns the next input as a *int*
`long`	`nextLong( )` returns the next input as a *long*
`float`	`nextFloat( )` returns the next input as a *long*
`double`	`nextDouble( )` returns the next input as a *double*
`boolean`	`nextBoolean( )` returns the next input as a *boolean*
`String`	`next( )` returns the next token in the input line as a *String*
`String`	`nextLine( )` returns the unread characters of the input line as a *String*

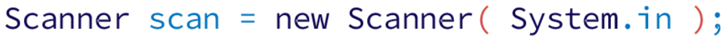
Description

The *Scanner* class is defined in the *java.util* package, so we need to include the following *import* statement:



```
import java.util.Scanner;
```

In order to use the *Scanner* class, we must first instantiate a *Scanner* object and associate it with a data source. We will use the *System.in* input stream, which by default is tied to the keyboard. Thus, our data source for input will be *System.in*. The following statement will instantiate a *Scanner* object named *scan* and associate the keyboard as its data source.

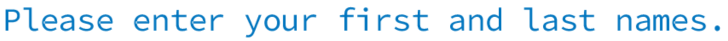


```
Scanner scan = new Scanner( System.in );
```

Once the *Scanner* object has been instantiated, we can use it to call any of the *next*. . . methods to input data from the keyboard. The specific *next*. . . method we call depends on the type of input we want from the user. Each of the *next*. . . methods returns a value from the input stream, that is, the value the user types into the keyboard. We need to assign the return value from the *next*. . . methods to a variable to complete the data input. Obviously, the data type of our variable must match or be compatible with the data type of the value returned by the *next*. . . method.

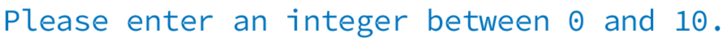
The *next*. . . methods just perform input. They do not tell the user what data to enter. Before calling any of the *next* methods, therefore, we need to prompt the user for the input we want. We can print a prompt using *System.out.print*, which is similar to using *System.out.println*, except that the cursor remains after the printed text rather than advancing to the next line.

When writing a prompt for user input, we should keep several things in mind. First, our prompt should be specific. If we want the user to enter his or her full name, then our prompt should say just that:



```
Please enter your first and last names.
```

If the input should fall within a range of values, then we should tell the user which values will be valid, for example:



```
Please enter an integer between 0 and 10.
```

We should also keep in mind that users are typically not programmers. It’s important to phrase a prompt using language the user understands. Many times, programmers write a prompt from their point of view, as in this bad prompt:

```
Please enter a String:
```

Users don’t know, and don’t care, about *Strings* or any other data types, for that matter. Users want to know only what they need to enter to get the program to do its job.

When our prompts are clear and specific, the user makes fewer errors and therefore feels more comfortable using our program.

SOFTWARE ENGINEERING TIP

Provide the user with clear prompts for input. Prompts should be phrased using words the user understands and should describe the data requested and any restrictions on input values.

Line 13 of **Example 3.9** prompts the user to enter his or her first name. Line 14 captures the user input and assigns the word entered by the user to the *String* variable *firstName*, which is printed in line 15. Similarly, line 17 prompts for the user’s age, line 18 captures the integer entered by the user and assigns it to the *int* variable *age*, and line 19 outputs the value of *age*. Reading other primitive data types follows the same pattern. Line 21 prompts for the user’s grade point average (a *float* value). Line 22 captures the number entered by the user and assigns it to the *float* variable *gpa*, and line 23 outputs the value of *gpa*.

Example 3.9 Reading from the Keyboard Using *Scanner*

```
/* A demonstration of reading from the keyboard using Scanner
   Anderson, Franceschi
*/

import java.util.Scanner;

public class DataInput
{
  public static void main( String [ ] args )
  {
    Scanner scan = new Scanner( System.in );

    System.out.print( "Enter your first name > " );
    String firstName = scan.next( );
    System.out.println( "Your name is " + firstName );

    System.out.print( "\nEnter your age as an integer > " );
    int age = scan.nextInt( );
    System.out.println( "Your age is " + age );

    System.out.print( "\nEnter your GPA > " );
    float gpa = scan.nextFloat( );
    System.out.println( "Your GPA is " + gpa );
  }
}
```

Description

When this program executes, the prompt is printed on the console and the cursor remains at the end of the prompt. **Figure 3.14** shows the output when these statements are executed and the user enters Syed, presses *Enter*, enters 21, presses *Enter*, and enters 3.875, and presses *Enter* again.

Figure 3.14 Data Input with Example 3.9

Description

SOFTWARE ENGINEERING TIP

End your prompts with some indication that input is expected, and include a trailing space for better readability.

The methods *nextByte, nextShort, nextLong, nextDouble*, and *nextBoolean* can be used with the same pattern as *next, nextInt*, and *nextFloat*.

Note that we end our prompt with a space, an angle bracket, and another space. The angle bracket indicates that we are waiting for input, and the spaces separate the prompt from the input. Without the trailing space, the user's input would immediately follow the prompt, which is more difficult to read, as we show in **Figure 3.15**.

Figure 3.15 Prompt and Input Running Together

As you review Table 3.7, you may notice that the *Scanner* class does not provide a method for reading a single character. To do this, we can use the *next* method, which returns a *String*, then extract the first character from the *String* using the *charAt(0)* method call, as shown in **Example 3.10**. Line 14 inputs a *String* from the user and assigns it to the *String* variable *initialS*, then line 15 assigns the first character of *initialS* to the *char* variable *initial*; *initial* is then printed at line 16 as shown in **Figure 3.16**.

Example 3.10 Using *Scanner* for Character Input

```
 1 /* A demonstration of how to get character input using Scanne
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6
 7 public class CharacterInput
 8 {
 9   public static void main( String [ ] args )
10   {
11     Scanner scan = new Scanner( System.in );
12
13     System.out.print( "Enter your middle initial > " );
14     String initialS = scan.next( );
15     char initial = initialS.charAt( 0 );
16     System.out.println( "Your middle initial is " + initial )
17   }
18 }
```

Description

Figure 3.16 Output of Example 3.10

Description

A *Scanner* object divides its input into sequences of characters called **tokens**, using **delimiters**. The default delimiters are the standard **white space** characters, which among others include the space, tab, and *newline* characters. The complete set of Java white space characters is shown in **Table 3.8**.

Java White Space Characters

Character	**Unicode Equivalents**
space	`\u00A0, \u2007, \u202F`
tab	`\u0009, \u000B`
line feed	`\u000A`
form feed	`\u000C`
carriage return	`\u000D`
file, group, unit, and record separators	`\u001C, \u001D, \u001E, \u001F`

Description

By default, when a *Scanner* object tokenizes the input, it skips leading white space, then builds a token composed of all subsequent characters until it encounters another delimiter. Thus, if we have this code,

```
System.out.print( "Enter your age as an integer > " );
int age = scan.nextInt( );
```

Description

and the user types, for example, three spaces and a tab, *21*, and a newline:

```
<space><space><space><tab>21<newline>
```

Description

then the *Scanner* object skips the three spaces and the tab, starts building a token with the character *2*, then adds the character *1* to the token, and stops building the token when it encounters the *newline* character. Thus, *21* is the resulting token, which the *nextInt* method returns and which is then assigned to the *age* variable.

An input line can contain more than one token. For example, if we prompt the user for his or her name and age, and the user enters the following line, then presses *Enter*:

```
<tab>Jon<space>Olsen,<space>21<space>
```

Description

then, the leading white space is skipped and the *Scanner* object creates three tokens:

- *Jon*
- *Olsen,*
- *21*

Note that commas are not white space, so the comma is actually part of the second token. To input these three tokens, our program would use two calls to the *next* method to retrieve the two *String* tokens and a call to *nextInt* to retrieve the age.

To capture a complete line of input from the user, we use the method *nextLine*. Unlike other *Scanner* methods, the *nextLine* method returns a *String* containing any leading and trailing white space, except for the *newline* character, which is the *nextLine* method's only delimiter. **Example 3.11** shows how *nextLine* can be used in a program. **Figure 3.17** shows a sample run of the program with the user entering data.

Example 3.11 Using the *nextLine* Method

```
/* A demonstration of using Scanner's nextLine method
   Anderson, Franceschi
*/

import java.util.Scanner;

public class InputALine
{
  public static void main( String [ ] args )
  {
    Scanner scan = new Scanner( System.in );

    System.out.print( "Enter a sentence > " );
    String sentence = scan.nextLine( );
    System.out.println( "You said: \"" + sentence + "\"" );
  }
}
```

Description

Figure 3.17 Output of Example 3.11

Description

REFERENCE POINT

You can read more about the *Scanner* class on Oracle's Java website: *www.oracle.com/technetwork/java*.

If the user's input (that is, the next token) is not compatible with the data type of the *next*. . . method call, then an *InputMismatchException* is generated and the program stops. **Figure 3.18** demonstrates Example 3.9 when the program calls the *nextInt* method and the user enters a letter, rather than an integer. It is possible to avoid this exception, and we show how when we discuss looping.

Figure 3.18 An Exception When Input Is Not the Expected Data Type

Description

If the user doesn't type anything when prompted, or if the user types some characters but doesn't press *Enter*, the program will simply wait until the user does press *Enter*.

with these end-of-chapter questions

3.18.1	Multiple Choice Exercises Questions 1, 11
3.18.2	Reading and Understanding Code Questions 14, 15, 16
3.18.3	Fill In the Code Questions 24, 25, 26, 27
3.18.4	Identifying Errors in Code Questions 36, 37, 38, 39, 43
3.18.5	Debugging Area Questions 45, 49
3.18.6	Write a Short Program Questions 50, 51, 52, 55, 56

3.11 Calling *Static* Methods and Using *Static* Class Variables

Classes can also define ***static* methods**, which can be called without instantiating an object. These are also called **class methods**. The API of these methods has the keyword *static* before the return type:

```
static dataType methodName( arg1, arg2, . . . )
```

One reason a class may define *static* methods is to provide some quick, one-time functionality without requiring the client to instantiate an object. For example, as part of a larger calculation, we might want to calculate the square root of a number. Creating an object just to calculate a square root is a waste of memory and processor time. Fortunately, the *Math* class provides a *static* method that finds the square root of a number.

Class, or *static*, methods are invoked using the class name, rather than an object reference, as in the following syntax:

```
ClassName.staticMethodName( argumentList );
```

For example, in this statement:

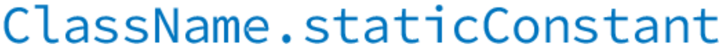
```
absValue = Math.abs( someNumber );
```

the class name is *Math*, and the *static* method is *abs*, which returns the absolute value of the argument (*someNumber*). We use the class name rather than an object reference, because *static* methods can be called without instantiating an object. Later in this chapter, we will explore some *static* methods of the *Math* class in greater detail.

Because *static* methods can be called without an object being instantiated, *static* methods cannot access the instance variables of the class (because instance variables are object data and exist only after an object has been instantiated). *Static* methods can access ***static* data**, however, and classes often declare *static* data to be used with *static* methods. *Static* data belong to the class, rather than to a particular object, or instance, of the class.

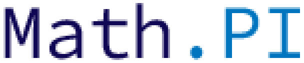
A common use of *static* class variables is to define constants for commonly used values or for parameters for the *static* class methods. For example, the *Math* class defines two *static* constants to facilitate the use of pi and the natural logarithm, e.

Like *static* methods, *static* constants are also accessed using the class name and dot operator, as in this syntax:



```
ClassName.staticConstant
```

Thus, the *static* constant representing the value of pi can be accessed this way:



```
Math.PI
```

At first, this may appear to go against our earlier discussion of encapsulation and the restrictions on clients directly accessing object data. Remember we said that the client needed to use accessor methods (getters) and mutator methods (setters) to access object data. The reasoning behind encapsulation is to protect the object data from corruption by the client. However, in this case, the *static* data is constant, so the client is unable to change it. For the client, directly accessing the class constant is easier and faster than calling a method.

```
Combine the arguments using concatenation
A double: 23.7, and an int: 78

Java is case sensitive: a is different from A

Create a variable and print its value
The grade is 3.81
```

3.12 Using *System.in* and *System.out*

In order to print program output to the screen, we have been using statements like

```
System.out.println( "The value of b is " + b );
```

and

```
System.out.print( "Enter your first name > " );
```

And to instantiate a *Scanner* object, we used this statement:

```
Scanner scan = new Scanner( System.in );
```

It is now time to look at these statements in depth and understand them completely.

System is an existing Java class in the *java.lang* package. One of its fields is a *static* constant, *out*, which represents the Java console by default. Another of its fields is a *static* constant, *in*, which represents the keyboard by default. Because *in* and *out* are *static*, we refer to them using the class name, *System*, and the dot notation:

```
System.out
System.in
```

Table 3.9 shows these *static* constants as well as the *static* *exit* method, which can be used to terminate a program. Calling *exit* at the end of a program is optional. After the last instruction is executed, the program will end in any case. However, the *exit* method of the *System* class can be useful if we want to stop execution at a place other than the usual end of the program.

***Static* Constants of the *System* Class and the *exit* Method**

Constant	**Value**
`in`	*static* constant that represents the standard input stream, by default the keyboard
`out`	*static* constant that represents the standard output stream, by default the Java console

A Useful *System* Method

Return value	**Method name and argument list**
`void`	`exit( int exitStatus )` *static* method that terminates the Java Virtual Machine. A value of 0 for *exitStatus* indicates a normal termination. Any other values indicate an abnormal termination and are used to signal that the program ended because an error occurred.

Description

System.out is an object of the *PrintStream* class, which is also an existing Java class; it can be found in the *java.io* package. The *out* object refers to the **standard output device**, which by default is the Java console.

The methods *print* and *println* belong to the *PrintStream* class and take arguments of any primitive type, a *String*, or an object reference. The only difference between *print* and *println* is that *println* will also print a *newline* character after it writes the output. **Table 3.10** shows some methods of the *PrintStream* class, which can be used with *System.out*.

PrintStream Methods for Use with *System.out*

	Useful *PrintStream* Methods
Return value	**Method name and argument list**
`void`	`print( argument )` prints *argument* to the standard output device. The argument can be any primitive data type, a *String* object, or another object reference.
`void`	`println( argument )` prints *argument* to the standard output device, then prints a *newline* character. The argument can be any primitive data type, a *String*, or another object reference.
`void`	`println( )` prints a *newline* character. This method is useful for skipping a line in the program's output.

Description

Example 3.12 demonstrates various ways to use the *print* and *println* methods:

Example 3.12 Demonstrating the *print* and *println* Methods

```
 1 /* Testing the print and println methods
 2    Anderson, Franceschi
 3 */
 4
 5 public class PrintDemo
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     System.out.println( "Combine the arguments using concatenation" );
10     System.out.println( "A double: " + 23.7 + ", and an int: " + 78 );
11
12     System.out.print( "\nJava is case sensitive: " );
13     System.out.println( 'a' + " is different from " + 'A' );
14
15     System.out.println( "\nCreate a variable and print its value" );
16     String s = new String( "The grade is" );
17     double grade = 3.81;
18     System.out.println( s + " " + grade );
19
20     System.exit( 0 ); // optional
21   }
22 }
```

Description

Lines 10 and 13 show how *print* or *println* can be used with various data types such as *double, int*, and *char*. Variables and expressions can also be used instead of literals, as shown in line 18, where two *Strings* and the *double* variable *grade* are output.

REFERENCE POINT

You can read more about the *System* and *PrintStream* classes on Oracle's Java website: *www.oracle.com/technetwork/java*.

The output of Example 3.12 is shown in **Figure 3.19**. We terminate the program by calling the *exit* method of the *System* class.

Figure 3.19 The Output from Example 3.12

Description

```
The value of e is 2.718281828459045
The value of pi is 3.141592653589793

The absolute value of 6.7 is 6.7

The absolute value of -6.7 is 6.7
```

```
The log of 5 is 1.6094379124341003

The square root of 9 is 3.0

4 to the power 3 is 64.0

43.5 to the power 3.4 is 372274.65827529586
```

```
23.4 rounded is 23

23.49 rounded is 23

23.5 rounded is 24

23.51 rounded is 24

23.6 rounded is 24
```

```
The smaller of 8 and 2 is 2

The larger of 8 and 2 is 8

The smallest of 8, 5, and 12 is 5
```

3.13 The *Math* Class

The *Math* class is also part of the *java.lang* package. As such, it is automatically available to any Java program; we do not need to use the *import* statement. The *Math* class provides two *static* constants (*E* and *PI*), as well as a number of *static* methods that save the programmer from writing some complex mathematical code.

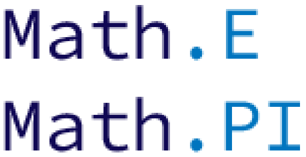
The two constants, *E* and *PI*, are both *doubles* and represent, respectively, ***e*** (the base of the natural logarithm, i.e., log e = 1) and ***pi***, the ratio of the circumference of a circle to its diameter. Approximate values of *e* and *pi*, as we know them, are 2.78 and 3.14, respectively. These constants are shown in **Table 3.11**.

***Static* Constants of the *Math* Class**

Constant	**Value**
`E`	*e*, the base of the natural logarithm
`PI`	*pi*, the ratio of the circumference of a circle to its diameter

Description

Because *E* and *PI* are *static* data members of the *Math* class, they are referenced using the name of the *Math* class and the dot notation as follows:



```
Math.E
Math.PI
```

Description

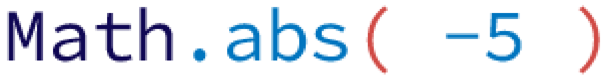
Useful methods of the *Math* class are shown in **Table 3.12**. All the methods of the *Math* class are *static*; so they are called using the class name, *Math*, and the dot notation as follows:

Useful Methods of the *Math* Class

Math Class Method Summary

Return value	Method name and argument list
`dataTypeOfArg`	`abs( arg )` *static* method that returns the absolute value of the argument *arg,* which can be a *double, float, int,* or *long*.
`double`	`log( double arg )` *static* method that returns the natural logarithm (in base *e*) of its argument, *arg*. For example, log(1) returns 0 and log(*Math.E)* returns 1.
`dataTypeOfArgs`	`min( argA, argB )` *static* method that returns the smaller of the two arguments. The arguments can be *doubles, floats, ints,* or *longs*.
`dataTypeOfArgs`	`max( argA, argB )` *static* method that returns the larger of the two arguments. The arguments can be *doubles, floats, ints,* or *longs*.
`double`	`pow( double base, double exp` *static* method that returns the value of *base* raised to the *exp* power.
`long`	`round( double arg )` *static* method that returns the closest integer to its argument, *arg.*
`double`	`sqrt( double arg )` *static* method that returns the positive square root of *arg*.

Description



```
Math.abs( -5 )
```

Description

Example 3.13 demonstrates how the *Math* constants and the *abs* method can be used in a Java program. In lines 9 and 10, we print the values of *e* and *pi* using the *static* constants of the *Math* class. Then in lines 12 and 15, we call the *abs* method, which returns the absolute value of its argument.

Example 3.13 *Math* Class Constants and the *abs* Method

```
 1 /* A demonstration of the Math class methods and constants
 2    Anderson, Franceschi
 3 */
 4
 5 public class MathConstants
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     System.out.println( "The value of e is " + Math.E );
10     System.out.println( "The value of pi is " + Math.PI );
11
12     double d1 = Math.abs( 6.7 ); // d1 will be assigned 6.7
13     System.out.println( "\nThe absolute value of 6.7 is " + d1 );
14
15     double d2 = Math.abs( -6.7 ); // d2 will be assigned 6.7
16     System.out.println( "\nThe absolute value of -6.7 is " + d2 );
17   }
18 }
```

Description

We then print the results in lines 13 and 16. The output of Example 3.13 is shown in Figure 3.20.

Figure 3.20 Output from Example 3.13

Description

The operation and usefulness of most *Math* class methods are obvious. But several methods—*pow*, *round*, and *min/max*—require a little explanation.

The *pow* Method

Example 3.14 demonstrates how some of these *Math* methods can be used in a Java program.

Example 3.14 A Demonstration of Some *Math* Class Methods

```
 1 /* A demonstration of some Math class methods
 2    Anderson, Franceschi
 3 */
 4
 5 public class MathMethods
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     double d2 = Math.log( 5 );
10     System.out.println( "\nThe log of 5 is " + d2 );
11
12     double d4 = Math.sqrt( 9 );
13     System.out.println( "\nThe square root of 9 is " + d4 );
14
15     double fourCubed = Math.pow( 4, 3 );
16     System.out.println( "\n4 to the power 3 is " + fourCubed );
17
18     double bigNumber = Math.pow( 43.5, 3.4 );
19     System.out.println( "\n43.5 to the power 3.4 is " + bigNumber );
20   }
21 }
```

Description

The *Math* class provides the *pow* method for raising a number to a power. The *pow* method takes two arguments; the first is the base and the second is the exponent.

Although the argument list for the *pow* method specifies that the base and the exponent are both *doubles*, we can, in fact, send arguments of any numeric type to the *pow* method because all numeric types can be promoted to a *double*. No matter what type the arguments are, however, the return value is always a *double*. Thus, when line 15 calls the *pow* method with two integer arguments, the value of *fourCubed* will be 64.0. If we prefer that the return value be 64, we can cast the return value to an *int*.

Line 18 shows how to use the *pow* method with arguments of type *double*. The output of Example 3.14 is shown in **Figure 3.21**.

Figure 3.21 Output from Example 3.14

Description

The *round* Method

The *round* method converts a *double* to its nearest integer using these rules:

- Any fractional part .0 to .4 is rounded down.
- Any fractional part .5 and above is rounded up.

Lines 9–13 in **Example 3.15** use the *round* method with various numbers. **Figure 3.22** shows the output.

Example 3.15 A Demonstration of the *Math round* Method

```
 1 /* A demonstration of the Math round method
 2    Anderson, Franceschi
 3 */
 4
 5 public class MathRounding
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     System.out.println( "23.4 rounded is " + Math.round( 23.4 ) );
10     System.out.println( "23.49 rounded is " + Math.round( 23.49 ) );
11     System.out.println( "23.5 rounded is " + Math.round( 23.5 ) );
12     System.out.println( "23.51 rounded is " + Math.round( 23.51 ) );
13     System.out.println( "23.6 rounded is " + Math.round( 23.6 ) );
14   }
15 }
```

Description

Figure 3.22 Output from Example 3.15

Description

The *min* and *max* Methods

The *min* and *max* methods return the smaller or larger of their two arguments, respectively. **Example 3.16** demonstrates how the *min* and *max* methods can be used in a Java program. **Figure 3.23** shows the output. Thus, the statement on line 9 of Example 3.16 will assign 2 to the *int* variable *smaller*. At line 12, a similar statement using the *max* method will assign 8 to the *int* variable *larger*.

Example 3.16 A Demonstration of the *min* and *max* Methods

```
 1 /* A demonstration of min and max Math class methods
 2    Anderson, Franceschi
 3 */
 4
 5 public class MathMinMaxMethods
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     int smaller = Math.min( 8, 2 );
10     System.out.println( "The smaller of 8 and 2 is " + smaller );
11
12     int larger = Math.max( 8, 2 );
13     System.out.println( "The larger of 8 and 2 is " + larger );
14
15     int a = 8, b = 5, c = 12;
16     int tempSmaller = Math.min( a, b ); // find smaller of a & b
17     int smallest = Math.min( tempSmaller, c ); // compare result to c
18     System.out.println( "The smallest of " + a + ", " + b + ", and "
19                          + c + " is " + smallest );
20   }
21 }
```

Description

Figure 3.23 Output from Example 3.16

Description

REFERENCE POINT

You can read more about the *Math* class on Oracle's Java website: *www.oracle.com/technetwork/java*.

The *min* method can also be used to compute the smallest of three variables. After declaring and initializing the three variables (*a, b*, and *c*) at line 15, we assign to a temporary variable named *tempSmaller* the smaller of the first two variables, *a* and *b*, at line 16. Then, at line 17, we compute the smaller of *tempSmaller* and the third variable, *c*, and assign that value to the variable *smallest*, which is output at lines 18 and 19.

The pattern for finding the largest of three numbers is similar, and we leave that as an exercise at the end of the chapter.

with these end-of-chapter questions

3.18.1	Multiple Choice Exercises Questions 6, 7, 8, 13
3.18.2	Reading and Understanding Code Questions 17, 18, 19, 20, 21, 22,

	23
3.18.3	Fill In the Code Questions 28, 29, 30, 31, 32, 34
3.18.4	Identifying Errors in Code Questions 40, 41, 42
3.18.5	Debugging Area Questions 46, 47, 48
3.18.6	Write a Short Program Questions 53, 54

CODE IN ACTION

To see a step-by-step illustration of how to instantiate an object and call both instance and *static* methods, look for this chapter's movie within the online resources. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

```
The winning percentage is 68%
The price is: $78.90
```

3.14 Formatting Output with the *NumberFormat* Class

Like the *DecimalFormat* class, the *NumberFormat* class can also be used to format numbers for output. The *NumberFormat* class, however, provides specialized *static* methods for creating objects specifically for formatting currency and percentages.

The *NumberFormat* class is part of the *java.text* package, so we need to include the following *import* statement at the top of our program.

```
import java.text.NumberFormat;
```

Description

The *static* methods of the *NumberFormat* class to format currency and percentages are shown in **Table 3.13**.

Useful Methods of the *NumberFormat* Class

	***NumberFormat* Method Summary**
Return value	**Method name and argument list**
`NumberFormat`	`getCurrencyInstance( )` *static* method that creates a format object for money
`NumberFormat`	`getPercentInstance( )` *static* method that creates a format object for percentages
`String`	`format( double number )`

	returns a *String* representation of *number* formatted as a currency or percentage, depending on the object used to call the method

Description

As we can see from the first two method headers, their return type is a *NumberFormat* object. These *static* methods, called **factory methods**, are used instead of constructors to create objects. Thus, instead of using the *new* keyword and a constructor, we will call one of these methods to create our formatting object.

The *getCurrencyInstance* method returns a formatting object that reflects the local currency. In the United States, that format is a leading dollar sign and is two digits to the right of the decimal place. The *getPercentInstance* method returns a formatting object for printing a fraction as a percentage by multiplying the fraction by 100, rounding to the nearest whole percent, and adding a percent sign (%).

We then use the *format* method from the *NumberFormat* class to display a value either as money or a percentage. The *format* method takes one argument, a *double,* which is the variable or value that we want to print; it returns the formatted version of the value as a *String* object, which we can then print.

Example 3.17 is a complete program illustrating how to use these three methods.

Example 3.17 Demonstrating the *NumberFormat* Class

```java
/* Demonstration of currency and percentage formatting
   using the NumberFormat class.
   Anderson, Franceschi
*/

// we need to import the NumberFormat class from java.text
import java.text.NumberFormat;

public class DemoNumberFormat
{
  public static void main( String [ ] args )
  {
    double winningPercentage = .675;
    double price = 78.9;

    // get a NumberFormat object for printing a percentage
    NumberFormat percentFormat = NumberFormat.getPercentInstance( );

    // call format method using the NumberFormat object
    System.out.print( "The winning percentage is " );
    System.out.println( percentFormat.format( winningPercentage ) );

    // get a NumberFormat object for printing currency
    NumberFormat priceFormat = NumberFormat.getCurrencyInstance( );

    // call format method using the NumberFormat object
    System.out.println( "\nThe price is: "
                        + priceFormat.format( price ) );
  }
}
```

Description

The output of this program is shown in **Figure 3.24**. As with *DecimalFormat*, the *NumberFormat format* method multiples the fraction to be formatted by 100. Note that the winning percentage is formatted as a rounded whole number. If we wanted to print the fraction .675 as 67.5%, we would need to use a *DecimalFormat* object.

Figure 3.24 Output from Example 3.17

Description

```
The int is 42
The Integer object is 42
Their sum is 84

The value of i1 is 76
The value of i2 is 76

The value of d1 is 58.32
The value of d2 is 58.32
```

```
Enter a character > d
The Unicode decimal value of d is: 100

d is a digit: false

d is a letter: true
d is uppercase: false
d is lowercase: true

d in uppercase is: D
d in lowercase is: d
```

3.15 The *Integer, Double, Character,* and Other Wrapper Classes

Most programs use a combination of primitive data types and objects. Some class methods, however, will accept only objects as arguments, so we need some way to convert a primitive data type into an object. Conversely, there are times when we need to convert an object into a primitive data type. For example, let’s say we have a GUI where we ask users to type their age into a text box or a dialog box. We expect the age to be an *int* value; however, text boxes and dialog boxes return their values as *Strings*. To perform any calculations on an age in our program, we will need to convert the value of that *String* object into an *int*.

For these situations, Java provides **wrapper classes**. A wrapper class “wraps” the value of a primitive type, such as *double* or *int*, into an object. These wrapper classes define an instance variable of that primitive data type, and also provide useful constants and methods for converting between the objects and the primitive data types. **Table 3.14** lists the wrapper classes for each primitive data type.

Wrapper Classes for Primitive Data Types

Primitive Data Type	Wrapper Class
double	Double
float	Float
long	Long
int	Integer
short	Short
byte	Byte
char	Character
boolean	Boolean

Description

All these classes are part of the *java.lang* package. So, the *import* statement is not needed in order to use wrapper classes in a program.

Java provides special support for converting between a primitive numeric type and its wrapper class. For example, we can simply assign an *int* variable to an *Integer* object reference. Java will automatically provide the conversion for us. This conversion is called **autoboxing**. In **Example 3.18**, the conversion is illustrated in lines 9 and 10. The *int* variable, *intPrimitive*, and the *Integer* object, *integerObject*, are output at lines 12 and 13 and have the same value (42). The output is shown in **Figure 3.25**.

Example 3.18 A Demonstration of the Wrapper Classes

```
 1 /* A demonstration of the Wrapper classes and methods
 2    Anderson, Franceschi
 3 */
 4
 5 public class DemoWrapper
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     int intPrimitive = 42;
10     Integer integerObject = intPrimitive;
11
12     System.out.println( "The int is " + intPrimitive );
13     System.out.println( "The Integer object is " + integerObject );
14
15     int sum = intPrimitive + integerObject;
16     System.out.println( "Their sum is " + sum );
17
18     int i1 = Integer.parseInt( "76" );    // convert "76" to an int
19     Integer i2 = Integer.valueOf( "76" ); // convert "76" to Integer
20     System.out.println( "\nThe value of i1 is " + i1 );
21     System.out.println( "The value of i2 is " + i2 );
22
23     double d1 = Double.parseDouble( "58.32" );
24     Double d2 = Double.valueOf( "58.32" );
25     System.out.println( "\nThe value of d1 is " + d1 );
26     System.out.println( "The value of d2 is " + d2 );
27   }
28 }
```

Description

Figure 3.25 Output from Example 3.18

Description

Similarly, when an *Integer* object is used as an *int*, Java also provides this conversion, which is called **unboxing**. Thus, when we use an *Integer* object in an arithmetic expression, the *int* value is automatically used. Line 15 of Example 3.18 uses the *Integer* object *integerObject* in an arithmetic expression, adding the *Integer* object to the *int* variable *intPrimitive*. As shown in Figure 3.25, the result is the same as if both operands were *int* variables.

REFERENCE POINT

You can read more about the wrapper classes on Oracle's Java website: *www.oracle.com/technetwork/java*.

Similar operations are possible using other numeric primitives and their associated wrapper classes.

In addition to automatic conversions between primitive types and wrapper objects, the *Integer* and *Double* classes provide methods, shown in **Table 3.15**, that allow us to convert *Strings* to numbers.

Methods of the *Integer* and *Double* Wrapper Classes

Useful Methods of the *Integer* Wrapper Class

Return value	Method name and argument list
`int`	`parseInt( String s )` *static* method that converts the *String s* to an *int* and returns that value
`Integer`	`valueOf( String s )` *static* method that converts the *String s* to an *Integer* object and returns that object

Useful Methods of the *Double* Wrapper Class

Return value	Method name and argument list
`double`	`parseDouble( String s )` *static* method that converts the *String s* to an *double* and returns that value
`Double`	`valueOf( String s )` *static* method that converts the *String s* to an *Double* object and returns that object

Description

The *parseInt*, *parseDouble*, and *valueOf* methods are *static* and are called using the *Integer* or *Double* class name and the dot notation. The *parse* methods convert a *String* to a primitive type, and the *valueOf* methods convert a *String* to a wrapper object. For example, line 18 of Example 3.18 converts the *String* “76” to the *int* value 76. Line 19 converts the *String* “76” to an equivalent *Integer* object.

Similarly, line 23 converts the *String* “58.32” to a *double*, and line 24 converts the same *String* to an equivalent *Double* object.

Because a *char* is a primitive type, we can’t call methods on a *char* variable. By wrapping a *char* variable into an object, however, the *Character* class provides some helpful methods for dealing with *chars*. Often, we may want to determine whether a character is a digit or a letter. For example, if we are checking whether a password is secure, we may want to determine whether the password has at least one digit, or we may want to change the case of a *char* variable. **Table 3.16** shows some useful methods of the *Character* class. All these methods are *static*, so we call them using the *Character* class name instead of an object reference. The first four methods test whether a *char* is a digit, a letter, lowercase, or uppercase. For each of these methods, we send the *char* variable as an argument and the method returns *true* or *false*. The next two methods, *toLowerCase* and *toUpperCase*, convert the *char* argument to lower- or uppercase if the *char* is a letter. We send the *char* variable to the method, and it returns a copy of the *char* variable after converting its case, if possible.

Useful Methods of the *Character* Wrapper Classes

Useful *static* Methods of the *Character* Wrapper Class

Return value	**Method name and argument list**
`boolean`	`isDigit( char c )` returns true if *c* is a character from ‘0’ to ‘9’ or a digit in other languages; false, otherwise
`boolean`	`isLetter( char c )` returns true if *c* is a letter; false, otherwise
`boolean`	`isLowerCase( char c )` returns true if *c* is a lowercase letter; false, otherwise
`boolean`	`isUpperCase( char c )`

	returns true if *c* is an uppercase letter; false, otherwise
`char`	`toLowerCase( char c )` returns the lowercase version of *c* if *c* is a letter; otherwise it returns *c* unchanged
`char`	`toUpperCase( char c )` returns the uppercase version of *c* if *c* is a letter; otherwise it returns *c* unchanged

Description

Example 3.19 demonstrates the use of these methods.

Example 3.19 Character Methods

```
 1 /*  Character Methods
 2     Anderson, Franceschi
 3 */
 4 import java.util.Scanner;
 5
 6 public class CharacterMethods
 7 {
 8   public static void main( String [ ] args )
 9   {
10     Scanner scan = new Scanner( System.in );
11
12     System.out.print( "Enter a character > " );
13     String input = scan.next( );
14     char c = input.charAt( 0 );
15
16     System.out.println( "The Unicode decimal value of " + c + " is: "
17                                  + ( int ) c );
18
19     System.out.println(  "\n" + c + " is a digit: "
20                                  + Character.isDigit( c ) );
21
22     System.out.println(  "\n" + c + " is a letter: "
23                                  + Character.isLetter( c ) );
24     System.out.println(  c + " is uppercase: "
25                                  + Character.isUpperCase( c ) );
26     System.out.println(  c + " is lowercase: "
27                                  + Character.isLowerCase( c ) );
28
29     System.out.println(  "\n" + c + " in uppercase is: "
30                                  + Character.toUpperCase( c ) );
31     System.out.println(  c + " in lowercase is: "
32                                  + Character.toLowerCase( c ) );
33   }
34 }
```

Description

Lines 12–14 prompt the user for a character to test or convert. Remember that *Scanner* does not have a *nextChar* method, so we need to use the *next* method to read a *String* and then extract the first character of the inputted *String* using the *charAt* method.

Lines 16–17 output the decimal Unicode value of the *char* variable by type casting it to an *int*.

At lines 19–20, we test whether the *char* is a digit by calling the *isDigit* method and output the returned value of *true* or *false*. Similarly, at lines 22–23, we test whether the *char* is a letter by calling the *isLetter* method and output the returned value of *true* or *false*.

At lines 24–27, we test whether the *char* is an uppercase or lowercase letter, again outputting the return value of *true* or *false*.

Finally, at lines 29–32, we attempt to convert the *char* to an uppercase letter and then to a lowercase letter. If *c* is not a letter, or if it is already an uppercase or lowercase letter, the methods return the value unchanged.

Figure 3.26 shows the output of the program when the user enters *d*.

Figure 3.26 Output from Example 3.19

Description

with these end-of-chapter questions

3.18.1	Multiple Choice Exercises Question 12
3.18.3	Fill In the Code Question 33
3.18.4	Identifying Errors in Code Question 35
3.18.5	Debugging Area Question 44
3.18.8	Technical Writing Questions 71, 72

```
Enter your first name > Esmerelda
Hello, Esmerelda
Your name has 9 letters.

In what year were you born? 2002
This year, you will be 18 years old.
So far, you have lived 22.9% of your expected life span.

I'm thinking of a number between 1 and 20
What is your guess? 15
The secret number was 2
Your guess was 13 away.
```

3.16 Programming Activity 2: Using Predefined Classes

In this Programming Activity, you will write a short program using some of the classes and methods discussed in this chapter. Your program will perform the following operations:

1. a. Prompt the user for his or her first name
   b. Print a message saying hello to the user
   c. Tell the user how many characters are in his or her name
2. a. Ask the user for the year of his or her birth
   b. Calculate and print the age the user will be this year
   c. Declare a constant for average life expectancy; set its value to 78.94
   d. Print a message that tells the user the percentage of his or her expected life lived so far formatted to one decimal place
3. a. Generate a random number between 1 and 20 inclusive
   b. Output a message telling the user that the program is thinking of a number between 1 and 20 and ask for a guess
   c. Output a message telling the user the number and how far away from the number the user's guess was

To complete this Programming Activity, copy the contents of this chapter's Programming Activity 2 folder in the supplied code accompanying this text. Open the *PracticeMethods.java* file and look for four sets of five asterisks (*****), where you will find instructions to write *import* statements and items 1, 2, and 3 for completing the Programming Activity.

Example 3.20 shows the *PracticeMethods.java* file, and **Figure 3.27** shows the output from a sample run after you have completed the Programming Activity. Your output might vary from that shown because we use 2020 as the current year for calculating age and because item 3 generates a random number.

Example 3.20 *PracticeMethods.java*

```
/* Chapter 3 Programming Activity 2
   Calling class methods
   Anderson, Franceschi
 */

// ***** 1. add your import statements here

public class PracticeMethods
{
  public static void main( String [ ] args )
  {
    //*****
    // 2.  a. Create a Scanner object to read from the keyboard.
    //     b. Prompt the user for their first name.
    //     c. Print a message that says hello to the user.
    //     d. Print a message that says how many letters
    //               are in the user's name.
    // Your code goes here


    //*****
    // 3.  a. Skip a line, then prompt the user for the year
    //            they were born.
    //     b. Declare a constant for the current year.
    //     c. Calculate and print the age the user will be this year.
    //     d. Declare a constant for average life expectancy,
    //            set its value to 78.94.
    //     e. Calculate and print the percentage
    //            of the user's expected life they've lived.
    //        Use the DecimalFormat class to format the percentage
    //            to one decimal place.
    // Your code goes here


    //*****
    // 4.  a. Generate a secret random integer between 1 and 20
    //     b. Skip a line, then ask the user for a guess.
    //     c. Print a message telling the user the secret number
    //         and how far from the number the user's guess was
    //         (hint: use Math.abs).
    // Your code goes here

  }
}
```

Description

Figure 3.27 Console Output from a Sample Run of Programming Activity 2

Description

DISCUSSION QUESTIONS

1. Which methods of the *Scanner* class did you choose for reading the user’s name and birth year? Explain your decisions.
2. How would you change your code to generate a random number between 10 and 20?

© Lois GoBe/Shutterstock.

CHAPTER REVIEW

3.17 Chapter Summary

- Object-oriented programming entails writing programs that use classes and objects. Using prewritten classes shortens development time and creates more reliable programs. Programs that use prewritten classes are called clients of the class.
- Benefits of object-oriented programming include encapsulation, reusability, and reliability.
- Classes consist of data, plus instructions that operate on that data. Objects of a class are created using the class as a template. Creating an object is called instantiating an object, and the object is an instance of the class. The *new* keyword is used to instantiate an object.
- The object reference is the variable name for an object and points to the data of the object.
- The data of a class are called fields and consist of instance variables and *static* variables. The instructions of the class are called methods. Methods of a class get or set the values of the data or provide other services of the class.
- The name of a method, along with its argument list and return value, is called the Application Programming Interface (API) of that method. Methods that are declared to be *public* can be called by any client of the class.

- By convention, class names in Java start with a capital letter. Method names, instance variables, and object names start with a lowercase letter. In all these names, embedded words begin with a capital letter.
- When our program makes a method call, control transfers to the instructions in the method until the method finishes executing. Then control is transferred back to our program.
- Instance methods are called using the object reference and the dot notation.
- A constructor is called when an object is instantiated. A constructor has the same name as the class and its job is to initialize the object's data. Classes can have multiple constructors.
- A method's data type is called the method's return type. If the data type is anything other than the keyword *void*, the method returns a value to the program. When a value-returning method finishes executing, its return value replaces the method call in the expression.
- When an object reference is first declared, its value is *null*. Attempting to use a *null* object reference to call a method generates an error.
- The garbage collector runs occasionally and deletes objects that have no object references pointing to them.
- Java packages are groups of classes arranged according to functionality. Classes in the *java.lang* packages are automatically available to Java programs. Other classes need to be imported.
- The *String* class can be used to create objects consisting of a sequence of characters. *String* constructors accept *String* literals, *String* objects, or no argument, which creates an empty *String*. The *length* method returns the number of characters in the *String* object. The *toUpperCase* and *toLowerCase* methods return a *String* in upper or lower case. The *charAt* method extracts a character from a *String*, while the *substring* method extracts a *String* from a *String*. The *indexOf* method searches a *String* for a character or substring.
- The *DecimalFormat* class, in the *java.text* package,

provides methods to format numeric output. For example, we can specify the number of digits to display after the decimal point or add dollar signs and percentage signs (%).

- The *Random* class, in the *java.util* package, provides methods to generate random numbers.
- The *Scanner* class, in the *java.util* package, provides methods for reading input from the keyboard. Methods are provided for reading primitive data types and *Strings*.
- When prompting the user for input, phrase the prompt in language the user understands. Describe the data requested and any restrictions on valid input values.
- *Static* methods, also called class methods, can be called without instantiating an object. *Static* methods can access only the *static* data of a class.
- *Static* methods are called using the class name and the dot notation.
- *System.out.println* prints primitive data types or a *String* to the Java console and adds a *newline* character. *System.out.println* with no argument skips a line. *System.out.print* prints the same data types to the Java console, but does not add a *newline.* Classes provide a *toString* method to convert objects to a *String* in order to be printed.
- The *Math* class provides *static* constants *PI* and *E* and *static* methods to perform common mathematical calculations, such as finding the maximum or minimum of two numbers, rounding values, and raising a number to a power.
- The *NumberFormat* class, in the *java.text* package, provides *static* methods for creating objects to format numeric output as currency or a percentage.
- Wrapper classes provide an object interface for a primitive data type. The *Integer* and *Double* wrapper classes provide *static* methods for converting between *ints* and *doubles* and *Strings*.
- The *Character* wrapper class provides methods for testing whether a character is a digit or a letter, and for converting letters to upper- or lowercase.

3.18 Exercises, Problems, and Projects

3.18.1 Multiple Choice Exercises

1. If you want to use an existing class from the Java Class Library in your program, what keyword should you use?

    - `use`
    - `import`
    - `export`
    - `include`

6. A constructor has the same name as the class name.

    - true
    - false

9. A given class can have more than one constructor.

    - true
    - false

12. What is the keyword used to instantiate an object in Java?

    - `make`
    - `construct`
    - `new`
    - `static`

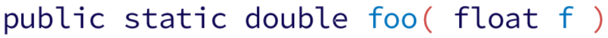
17. In a given class named *Quiz*, there can be only one method with the name *Quiz*.

- true
- false

20. A *static* method is

- a class method.
- an instance method.

23. In the *Quiz* class, the *foo* method has the following API:

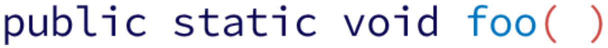


```
public static double foo( float f )
```

Description
What can you say about *foo?*

- It is an instance method.
- It is a class field.
- It is a class method.
- It is an instance variable.

28. In the *Quiz* class, the *foo* method has the following API:

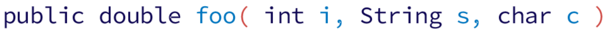


```
public static void foo( )
```

Description
How would you call that method?

- `Quiz.foo( );`
- `Quiz.foo( 8 );`
- `Quiz( foo( ) );`

32. In the *Quiz* class, the *foo* method has the following API:



```
public double foo( int i, String s, char c )
```

Description
How many arguments does *foo* take?

- 0
- 1
- 2
- 3

37. In the *Quiz* class, the *foo* method has the following API:

```
public double foo( int i, String s, char c )
```

Description
What is the return type of method *foo*?

- `double`
- `int`
- `char`
- `String`

42. *String* is a primitive data type in Java.

- true
- false

45. Which one of the following is not an existing wrapper class?

- `Integer`
- `Char`
- `Float`
- `Double`

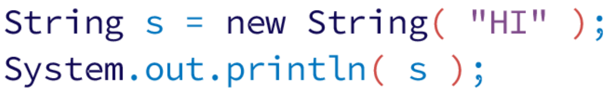
50. What is the proper way of accessing the constant *E* of the *Math* class?

- `Math.E( )x;`

- `Math.E;`
- `E;`
- `Math( E );`

3.18.2 Reading and Understanding Code

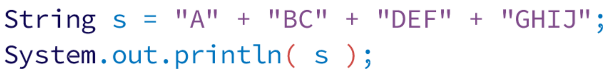
1. What is the output of this code sequence?

   ```
   String s = new String( "HI" );
   System.out.println( s );
   ```

   Description

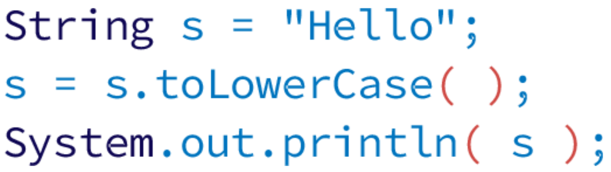
2. What is the output of this code sequence?

   ```
   String s = "A" + "BC" + "DEF" + "GHIJ";
   System.out.println( s );
   ```

   Description

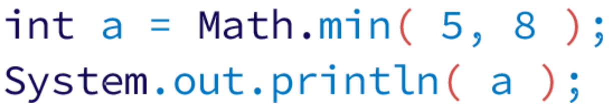
3. What is the output of this code sequence?

   ```
   String s = "Hello";
   s = s.toLowerCase( );
   System.out.println( s );
   ```

   Description

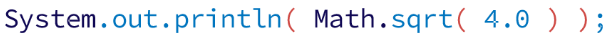
4. What is the output of this code sequence?

   ```
   int a = Math.min( 5, 8 );
   System.out.println( a );
   ```

   Description

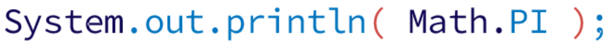
5. What is the output of this code sequence?

   ```
   System.out.println( Math.sqrt( 4.0 ) );
   ```

Description

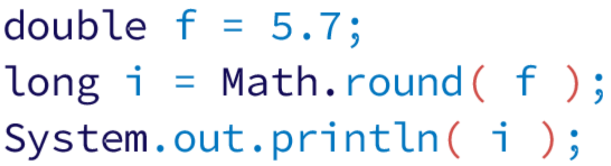
6. What is the output of this code sequence? (You will need to actually compile this code and run it in order to have the correct output.)



```
System.out.println( Math.PI );
```

Description

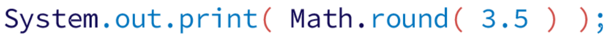
7. What is the output of this code sequence?



```
double f = 5.7;
long i = Math.round( f );
System.out.println( i );
```

Description

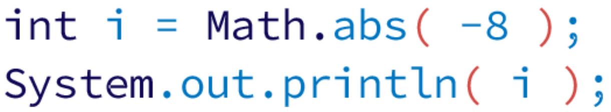
8. What is the output of this code sequence?



```
System.out.print( Math.round( 3.5 ) );
```

Description

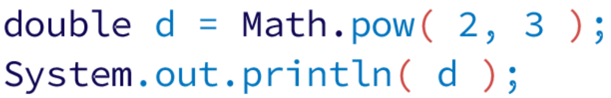
9. What is the output of this code sequence?



```
int i = Math.abs( -8 );
System.out.println( i );
```

Description

10. What is the output of this code sequence?

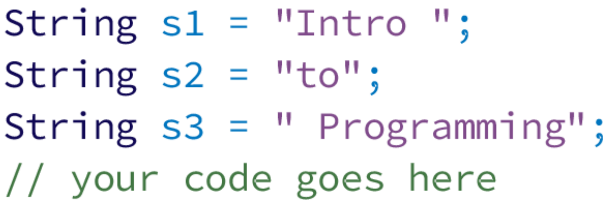


```
double d = Math.pow( 2, 3 );
System.out.println( d );
```

Description

3.18.3 Fill In the Code

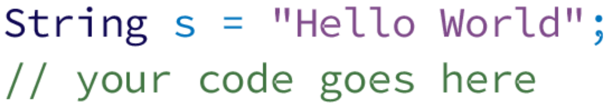
1. This code concatenates the three *Strings* “Intro ”, “to”, “ Programming” and outputs the resulting *String*. (Your output should be *Intro to Programming*.)

   ```
   String s1 = "Intro ";
   String s2 = "to";
   String s3 = " Programming";
   // your code goes here
   ```

   Description

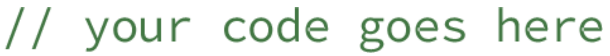
2. This code prints the number of characters in the *String* “Hello World”.

   ```
   String s = "Hello World";
   // your code goes here
   ```

   Description

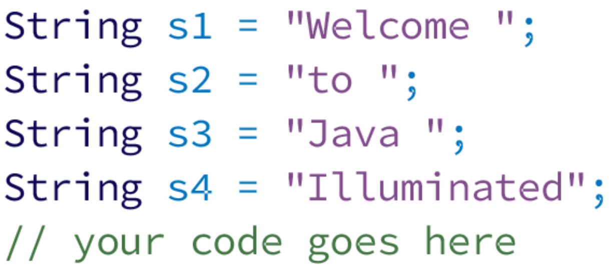
3. This code prompts the user for a *String*, then prints the *String* and the number of characters in it.

   ```
   // your code goes here
   ```

   Description

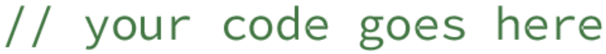
4. This code uses only a single line *System.out.println*. . . statement in order to print “Welcome to Java Illuminated” on one line using the following variables and the *String* concatenation operator:



```
String s1 = "Welcome ";
String s2 = "to ";
String s3 = "Java ";
String s4 = "Illuminated";
// your code goes here
```

Description

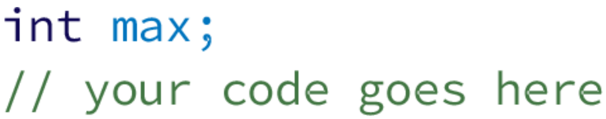
5. This code uses exactly four *System.out.print* statements in order to print “Welcome to Java Illuminated” on the same output line.



```
// your code goes here
```

Description

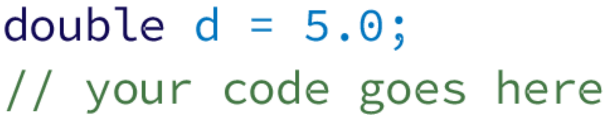
6. This code assigns the maximum of the values 3 and 5 to the *int* variable *max* and outputs the result.



```
int max;
// your code goes here
```

Description

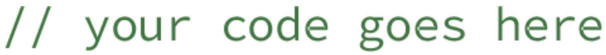
7. This code calculates the square root of 5 and outputs the result.



```
double d = 5.0;
// your code goes here
```

Description

8. This code asks the user for two integer values, then calculates the minimum of the two values and prints it.



```
// your code goes here
```

Description

9. This code asks the user for three integer values, then calculates the maximum of the three values and prints it.

```
// your code goes here
```

Description

10. This code prompts the user for a single character and prints “true” if the character is a letter and “false” if it is not a letter.

```
// your code goes here
```

Description

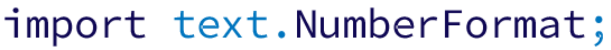
11. This code asks the user for a *double*, then prints the square of that number using the *pow* method of the *Math* class.

```
// your code goes here
```

Description

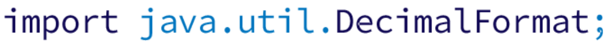
3.18.4 Identifying Errors in Code

1. Where is the error in this statement?



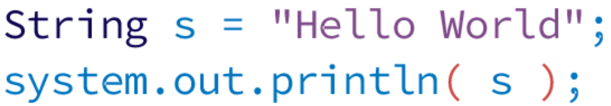
```
import text.NumberFormat;
```

2. Where is the error in this statement?



```
import java.util.DecimalFormat;
```

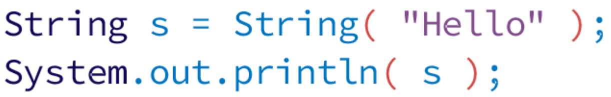
3. Where is the error in this code sequence?



```
String s = "Hello World";
system.out.println( s );
```

Description

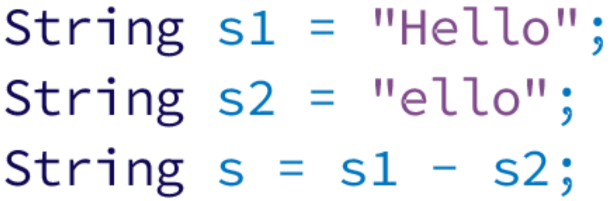
4. Where is the error in this code sequence?



```
String s = String( "Hello" );
System.out.println( s );
```

Description

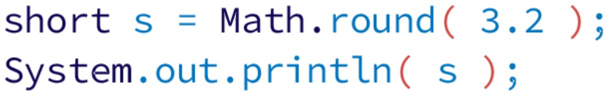
5. Where is the error in this code sequence?



```
String s1 = "Hello";
String s2 = "ello";
String s = s1 - s2;
```

Description

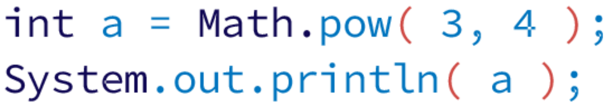
6. Where is the error in this code sequence?



```
short s = Math.round( 3.2 );
System.out.println( s );
```

Description

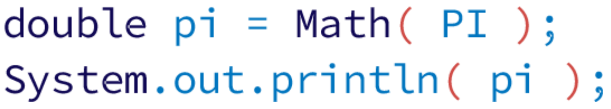
7. Where is the error in this code sequence?



```
int a = Math.pow( 3, 4 );
System.out.println( a );
```

Description

8. Where is the error in this code sequence?



```
double pi = Math( PI );
System.out.println( pi );
```

Description

9. Where is the error in this code sequence?

```
String s = 'H';
System.out.println( "s is " + s );
```

Description

3.18.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. You coded the following program in the file *Test.java:*

```
public class Test
{
  public static void main( String [ ] args )
  {
    int a = 6;
    NumberFormat nf = NumberFormat.getCurrencyInstance( );
  }
}
```

Description

When you compile, you get the following message:

```
Test.java: 6: error: cannot find symbol
    NumberFormat nf = NumberFormat.getCurrencyInstance( );
    ^
  symbol : class NumberFormat
  location: class Test
Test.java: 6: error: cannot find symbol
    NumberFormat nf = NumberFormat.getCurrencyInstance( );
                      ^
  symbol : variable NumberFormat
  location: class Test
2 errors
```

Description

Explain what the problem is and how to fix it.

2. You coded the following on lines 10–12 of class *Test.java:*

```
String s;                    // line 10
int l = s.length( );         // line 11
System.out.println( "length is " + l );      // line 12
```

Description

When you compile, you get the following message:

```
Test.java:11: error: variable s might not have been initialized.
    int l = s.length( );     // line 11
            ^
1 error
```

Description

Explain what the problem is and how to fix it.

3. You coded the following on lines 10 and 11 of class *Test.java:*

```
double d = math.sqrt( 6 );          // line 10
System.out.println( "d = " + d );   // line 11
```

Description

When you compile, you get the following message:

```
Test.java: 10: error: cannot find symbol
    double d = math.sqrt( 6 ); // line 10
               ^
    symbol : variable math
    location: class Test
1 error
```

Description

Explain what the problem is and how to fix it.

4. You coded the following on lines 10 and 11 of class *Test.java:*

```
double d = Math.PI( );             // line 10
System.out.println ( "d = " + d ); // line 11
```

Description

When you compile, you get the following message:

```
Test.java:10: error: cannot find symbol
    double d = Math.PI( );   // line 10
                   ^
    symbol : method PI( )
    location: class Math
1 error
```

Description

Explain what the problem is and how to fix it.

5. You coded the following on lines 10 and 11 of class *Test.java:*

```
double d = Math.e;                   // line 10
System.out.println( "d = " + d ); // line 11
```

Description

When you compile, you get the following message:

```
Test.java:10: error: cannot find symbol
    double d = Math.e;                      // line 10
                   ^
    symbol : variable e
    location: class Math
1 error
```

Description

Explain what the problem is and how to fix it.

6. You imported the *DecimalFormat* class and coded the following in the class *Test.java:*

```
double grade = .895;
DecimalFormat percent =
  new DecimalFormat( "#.0%" );

System.out.println( "Your grade is " + grade );
```

Description

The code compiles properly and runs, but the result is not what you expected. You expect this output:

```
Your grade is 89.5%
```

Description

But instead, the output is

```
Your grade is 0.895
```

Description

Explain what the problem is and how to fix it.

3.18.6 Write a Short Program

1. Write a program that reads two words representing passwords from the keyboard and outputs the number of characters in the smaller of the two. For example, if the two words are *open* and *sesame*, then the output should be *4*, the length of the shorter word, *open*.
2. Write a program that prompts the user for a domain name. Your program should then concatenate that name with *www.* and *.com* in order to form an Internet domain name and output the result. For instance, if the name entered by the user is *yahoo*, then the output will be *www.yahoo.com*.
3. Write a program that reads a word from the keyboard. Your program should output the word in uppercase letters only, output that word in lowercase letters only, and then, at the end, output the original word.
4. Write a program that generates two random numbers between 0 and 100 and prints the smaller of the two numbers.
5. Write a program that takes a *double* as an input, then computes and outputs the cube of that number using the *pow* method of the *Math* class.
6. Write a program that reads a file name from the keyboard. You should expect that the file name has one . (dot) character in it, separating the file name from the file extension. Retrieve the file extension and output it. For instance, if the user inputs *index.html*, you should output *html*; if the user inputs *MyClass.java*, you should output *java*.
7. Write a program that reads a full name (first name and last name) from the keyboard as a single line; you should expect the first name and the last name to be separated by a space. Retrieve and output the first and last names.

3.18.7 Programming Projects

1. Write a program that reads three integer values from the keyboard representing, respectively, a number of quarters, dimes, and nickels. Convert the total coin amount to dollars and output the result with a dollar notation.
2. Write a program that reads from the keyboard the radius of a circle. Calculate and output the area and the circumference of that circle. You can use the following formulas:

   $$\text{area} = \pi * \text{r}^2$$
   $$\text{circumference} = 2 * \pi * \text{r}$$

   Description
3. Write a program that generates five random integers between 60 and 100 and calculates the smallest of the five numbers.
4. Write a program that generates three random integers between 0 and 50, calculates the average, and prints the result to one decimal place.
5. Write a program that reads two integers from the keyboard: one representing the number of shots taken by a basketball player, the other representing the number of shots made by the same player. Calculate the shooting percentage and output it with the percent notation.
6. Write a program that reads three *double* numbers from the keyboard representing, respectively, the three coefficients *a*, *b*, and *c* of a quadratic equation. Solve the equation using the following formulas:

   $\text{x1} = (-b + \text{square root } (b^2 - 4ac))/(2a)$

   $\text{x2} = (-b - \text{square root } (b^2 - 4ac))/(2a)$

Description

Run your program on the following sample values:

$a = 1.0$, $b = 3.0$, $c = 2.0$
$a = 0.5$, $b = 0.5$, $c = 0.125$
$a = 1.0$, $b = 3.0$, $c = 10.0$

Description

Discuss the results for each program run, in particular what happens in the last case.

7. Write a program that reads two numbers from the keyboard representing, respectively, an investment and an interest rate (you will expect the user to enter a number such as .065 for the interest rate, representing a 6.5% interest rate). Your program should calculate and output (in $ notation) the future value of the investment in 5, 10, and 20 years using the following formula:

   $\text{future value} = \text{investment} * (1 + \text{interest rate})^{\text{year}}$

   We will assume that the interest rate is an annual rate and is compounded annually.

8. Write a program that reads from the keyboard the (x,y) coordinates for two points in the plane. You can assume that all numbers are integers. Using the *Point* class from the Java Class Library (you may need to look it up on the Web), instantiate two *Point* objects with your input data, then output the data for both *Point* objects.

9. Write a program that reads an email address. Assuming that the email address contains one at sign (@), extract and print the username and the domain name of the email address.
10. Write a program that reads a telephone number from the keyboard as a *String* of 10 digits. You should output that same telephone number formatted as (nnn) nnn-nnnn.
11. Write a program that reads a sentence from the keyboard. The sentence has been encrypted so that the message consists of only the first five characters with even-numbered indexes. All other characters should be discarded. Decrypt the sentence and output the result. For example, if the user inputs “Hiejlzl3ow”, your output should be *Hello*.
12. Write a program that reads a commercial website URL from the keyboard; you should expect that the URL starts with *www.* and ends with *.com*. Retrieve the name of the site

and output it. For instance, if the user inputs *www.yahoo.com*, you should output *yahoo*.

3.18.8 Technical Writing

1. At this point, we have written and debugged many examples of code. When you compile a Java program with the Java compiler, you get a list of all the errors in your code. Do you like the Java compiler? Do the error messages it displays when your code does not compile help you determine what’s wrong? How?
2. Computers, computer languages, and application programs existed before object-oriented programming. However, OOP has become an industry standard. Discuss the advantages of using OOP compared to using only basic data types in a program.
3. Explain and discuss a situation where you would use the method *parseInt* of the class *Integer*.
4. In addition to the basic data types (*int*, *float*, *char*, *boolean*, …), Java provides many prewritten classes, such as *Math*, *NumberFormat*, and *DecimalFormat*. Why is this an advantage? How does this impact the way a programmer approaches a programming problem in general?

3.18.9 Group Project (for a group of 1, 2, or 3 students)

1. Write a program that calculates a monthly mortgage payment; we will assume that the interest rate is compounded monthly.
   You will need to do the following:
    - Prompt the user for a *double* representing the annual

interest rate. For example, 3.5% would be entered as .035.

- Prompt the user for the number of years the mortgage will be held (typical input here is 10, 15, or 30).
- Prompt the user for a number representing the mortgage amount borrowed from the bank.
- Calculate the monthly payment using the following formulas:
  - Monthly payment = (mIR * M) / (1 − (1 / (1 + mIR)$^{(12*nOY)}$)), where:
    - mIR = monthly interest rate = annual interest rate / 12
    - nOY = number of years
    - M = mortgage amount
- Output a summary of the mortgage problem, as follows:
  - the annual interest rate in percent notation
  - the mortgage amount in dollars
  - the monthly payment in dollars, with only two significant digits after the decimal point
  - the total payment over the years, with only two significant digits after the decimal point
  - the overpayment, i.e., the difference between the total payment over the years and the mortgage amount, with only two significant digits after the decimal point
  - the overpayment as a percentage (in percent notation) of the mortgage amount

© Lois GoBe/Shutterstock.

CHAPTER 4

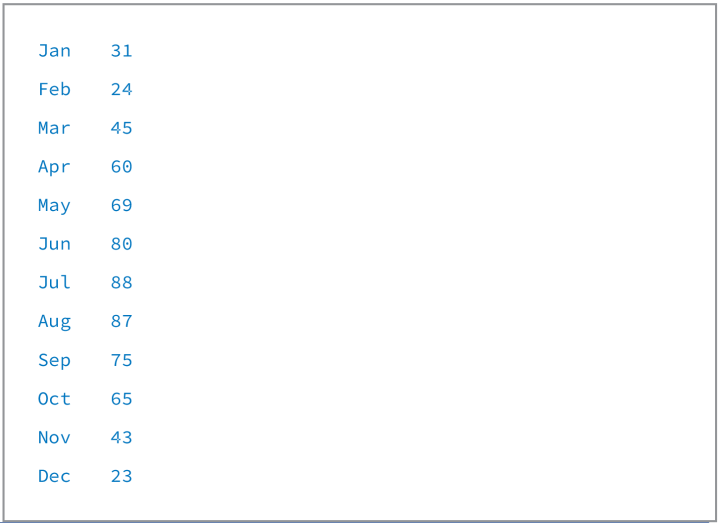
Introduction to Graphical Applications

CHAPTER CONTENTS

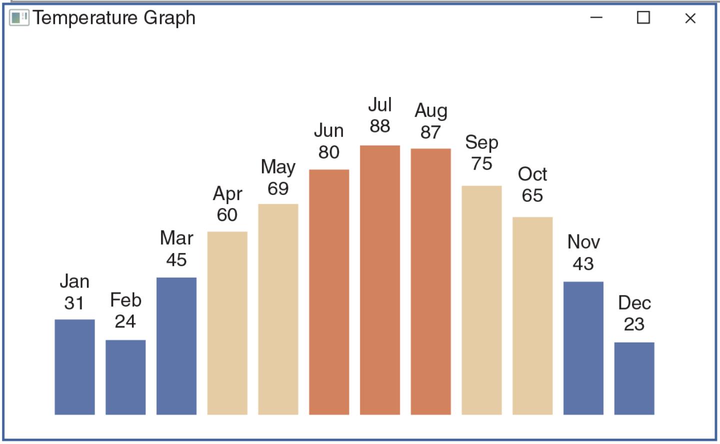
Introduction
4.1 JavaFX Application Structure
4.2 The Graphics Coordinate System and Color
4.3 Drawing Shapes and Text
4.4 Drawing Custom Shapes
4.5 **Programming Activity 1: Writing an Application with Graphics**
4.6 Chapter Summary
4.7 Exercises, Problems, and Projects

    4.7.1 Multiple Choice Exercises
    4.7.2 Reading and Understanding Code
    4.7.3 Fill In the Code
    4.7.4 Identifying Errors in Code
    4.7.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
    4.7.6 Write a Short Program
    4.7.7 Programming Projects
    4.7.8 Technical Writing

4.7.9 Group Project



```
Jan    31
Feb    24
Mar    45
Apr    60
May    69
Jun    80
Jul    88
Aug    87
Sep    75
Oct    65
Nov    43
Dec    23
```


Temperature Graph
Jan
31
Feb
24
Mar
45
Apr
60
May
69
Jun
80
Jul
88
Aug
87
Sep
75
Oct
65
Nov
43
Dec
23

Introduction

Graphical output is an integral part of many programs today. One compelling reason for using graphics in a program is the ability to present data in a format that is easy to comprehend. For example, our application could output average monthly temperatures as text, as shown in **Figure 4.1**.

Figure 4.1 Outputting Monthly Temperatures as Text

Description

Or we could produce the bar chart shown in **Figure 4.2**.

Figure 4.2 Bar Chart of Monthly Temperatures

Description

The bar chart presents the same information as the text output, but it adds a visual component that makes it easier to compare the monthly temperatures—for example, to find the highest or lowest temperature or to spot temperature trends throughout the year. The colors also add information, with the low temperatures shown in blue, the moderate temperatures in yellow, and the high temperatures in red.

In this chapter, we use JavaFX to write programs that produce graphical output.

4.1 JavaFX Application Structure

Over the years, Java has provided several incarnations of graphics systems. The first was Abstract Window Toolkit (AWT), which was subsequently replaced by Swing. Java's current system for building graphics applications is JavaFX. JavaFX makes it possible to create rich user interfaces with animations and special effects. In this chapter, we concentrate on JavaFX's features for creating graphical output for applications.

In order to use JavaFX, you need to download the JavaFX SDK. It can be downloaded as a bundle with the Java SDK, or separately if one has already downloaded the Java SDK. At the time of this edition, Oracle has open sourced JavaFX. It can be downloaded at *https://openjfx.io/*.

Oracle provides tutorial information on JavaFX at *http://docs.oracle.com/javafx/2/get_started/jfxpub-get_started.htm* and documentation for all the JavaFX classes is included with the latest Java documentation.

JavaFX uses a mixed performance/art metaphor. We start with a stage (a window) to which we add one or more scenes. To produce graphical output, we add a canvas to our scene, and we draw shapes and text on the canvas. When our application begins, the window opens and displays whatever we have drawn on the canvas.

REFERENCE POINT

For a tutorial on JavaFX, visit Oracle's website at *http://docs.oracle.com/javafx/2/get_started/jfxpub-get_started.htm*

We, the authors, are providing two classes to get you started. First, we provide a utility program, *JIGraphicsUtility.java*, to manage some of the common code needed to set up a JavaFX application. Second, we provide *ShellGraphicsApplication.java*, which provides a framework for adding your custom drawing code. The *JIGraphicsUtility.java* class, shown in **Example 4.1**, has one *static* method, *setUpGraphics*, which is designed to be called from our graphics application to set up our window title and size. We show the *JIGraphicsUtility* class here for completeness, but we do not need to make any modifications to this file. All we need to do is include this file in the same folder as our graphics application and call the *setUpGraphics* method from our application. The *setUpGraphics* method has the API shown in **Table 4.1**. In our graphics application, we use the *GraphicsContext* object returned from the *setUpGraphics* method to call methods that draw shapes and text on the window.

Example 4.1 *JIGraphicsUtility* Class

```java
/* JIGraphicsUtility
   Utility class for building JavaFX graphics applications
   Anderson, Franceschi
*/

import javafx.scene.Group;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.Scene;
import javafx.stage.Stage;

 public class JIGraphicsUtility
{
     public static GraphicsContext setUpGraphics( Stage stage,
                          String title, int height, int width )
     {
        stage.setTitle( title );
        Canvas canvas = new Canvas( height, width );
        GraphicsContext gc = canvas.getGraphicsContext2D( );
        Group root = new Group( canvas );
        stage.setScene( new Scene( root ) );
        stage.show( );
        return gc;
     }
}
```

Description

The *static setUpGraphics* Method

Return Value	**Method Name and Argument List**
`GraphicsContext`	`setUpGraphics( Stage`

	`stage, String title, int width, int height )` initializes a scene on the *stage*, sets the title of the window to *title*, and adds a canvas with the *width* and *height* specified; returns a *GraphicsContext* object

Description

Example 4.2 shows the *ShellGraphicsApplication* class. This class provides skeleton code for an application. We create a new application by modifying this code.

Example 4.2 *ShellGraphicsApplication.java*

```
 1 /* A Shell for creating graphics applications
 2    Anderson, Franceschi
 3 */
 4
 5 import javafx.application.Application;
 6 import javafx.scene.canvas.GraphicsContext;
 7 import javafx.stage.Stage;
 8
 9 public class ShellGraphicsApplication extends Application
10 {
11     @Override
12     public void start( Stage stage )
13     {
14         // set up window title and size
15         GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
16                   stage, "Shell Graphics Application", 700, 400 );
17
18         // your drawing code goes here
19     }
20
21     public static void main( String [ ] args )
22     {
23         launch( args );
24     }
25 }
```

Description

Lines 5–7 import the minimum classes needed for a JavaFX application. Notice that all the package names begin with *javafx*.

We define our application name in line 9. Here the name is *ShellGraphicsApplication*. For each new application we write, we will change this name to reflect the application’s purpose. The phrase “*extends Application*” means that we inherit basic functionality and methods from the JavaFX *Application* class.

One of the methods we inherit is the *start* method, which the JavaFX platform calls automatically when the application begins. It receives one parameter, *stage*, which represents the application’s window. We need to write our own version of the *start* method (lines 11–19), however, in which we put our method calls to draw shapes and text on the window. We use the *@Override* annotation in line 11 to indicate that our *start* method replaces the *start* method in the *Application* class.

Our first order of business is to call the *setUpGraphics* method (lines 14–16). Because *setUpGraphics* is a *static* method, we call it using the *JIGraphicsUtility* class name. Here we send as arguments the stage representing our window; a title for the window; and a size for the window, in this case a width of 700 pixels and a height of 400 pixels. For our applications, we can change the title to something appropriate and alter the window size as needed.

Line 18 indicates where we add code to create our graphics.

In lines 21–24 we define the *main* method. Its only job is to launch the JavaFX application by calling the *launch* method of the *Application* class, passing through any arguments that were sent to the *main* method. Because this code will be the same for every application, we place *main* at the end of the program.

Thus, to run a JavaFX graphics application, we alter the *ShellGraphicsApplication.java* class to add our custom graphics code, and we put the *JIGraphicsUtility.java* class and our custom class in the same folder. We then compile both classes and execute our custom class.

On the author’s computer, the JavaFX version is 17.0.1 and is installed in the *C:\Program Files\JavaFX* folder.

To compile a JavaFX program from the command line, type:

```
javac --module-path <path of JavaFX lib folder> --add-modules
javafx.controls ApplicationClassName.java
```

Description

On the author’s computer, it would be:

```
javac --module-path "C:\Program Files\JavaFX\javafx-sdk-17.0.1\lib"
--add-modules javafx.controls ApplicationClassName.java
```

Description

To compile the *ShellGraphicsApplication* class from the command line, it would be:

```
javac --module-path "C:\Program Files\JavaFX\javafx-sdk-17.0.1\lib"
--add-modules javafx.controls ShellGraphicsApplication.java
```

Description

After compilation, to run a JavaFX from the command line, type:

```
java --module-path <path of JavaFX lib folder> --add-modules
javafx.controls ApplicationClassName
```

Description

On the author's computer, it would be:

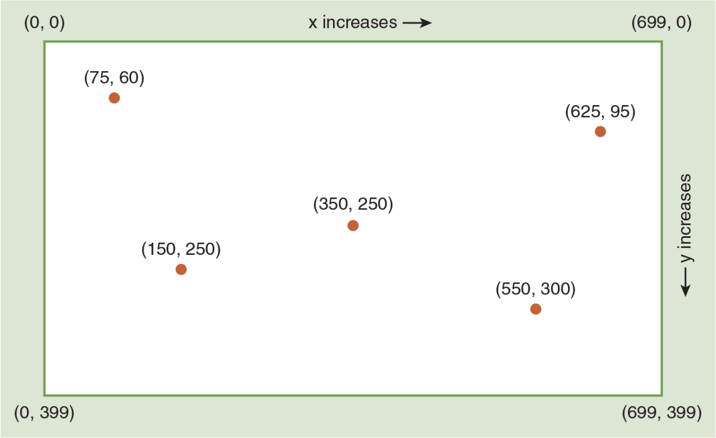
```
java --module-path "C:\Program Files\JavaFX\javafx-sdk-17.0.1\lib"
--add-modules javafx.controls ApplicationClassName
```

Description

To run the *ShellGraphicsApplication* from the command line, it would be:

```
java --module-path "C:\Program Files\JavaFX\javafx-sdk-17.0.1\lib"
--add-modules javafx.controls ShellGraphicsApplication
```

Description



4.2 The Graphics Coordinate System and Color

JavaFX's *GraphicsContext* class, in the *javafx.scene.canvas* package, provides methods to draw figures such as rectangles, circles, and lines; to set the colors for drawing; and to write text in a window.

To draw on the window, we specify the location where we want to put our shapes and the color we want the shapes to have.

A window is composed of pixels, which can be thought of as colored dots. A window with a width of 700 pixels and a height of 400 pixels has a total of 28,000 (700 * 400) pixels. Each pixel has a location expressed using an (*x*, *y*) coordinate system. The *x* coordinate specifies the horizontal position, beginning at 0 on the left side of the window and increasing as you move to the right. The *y* coordinate specifies the vertical position, starting at 0 at the top of the window and increasing as you move down. Thus, for a window that is 700 pixels wide and 400 pixels high, the coordinate (0, 0) corresponds to the upper-left corner; (699, 0) is the upper-right corner; (0, 399) is the lower-left corner; and (699, 399) is the lower-right corner. **Figure 4.3** shows a window with a few sample pixels and their (*x*, *y*) coordinates.

Figure 4.3 The *GraphicsContext* Coordinate System

Description

Each pixel also has a color. When a window opens, all pixels are set to the **background color**, which by default is white. Any drawing we perform is done in the **foreground colors**, which are the **stroke color** and the **fill color**. Most of the drawing methods have a stroke and a fill version, where the stroke version draws an outlined shape in the current stroke color and the fill version draws a solid shape in the current fill color. By default, both the stroke color and fill color are black, but we can set either to any color we prefer. The values set for the colors remain in effect until we set either the stroke color or the fill color to another color. For example, if we set the stroke color to blue and the fill color to red, then draw an outlined rectangle and a line, as well as a solid oval, the rectangle and line will be drawn in blue and the oval will be drawn in red. Then if we set the fill color to yellow and draw a solid rectangle, that rectangle will be drawn in yellow.

To set the foreground colors, we call the *setFill* or *setStroke* methods of the *GraphicsContext* class as shown in **Table 4.2**. These methods take as an argument, a *Paint* object. Because objects of the *Color* class are also *Paint* objects, we can send *Color* objects as arguments to the *setFill* and *setStroke* methods.

Methods of the *GraphicsContext* Class to Set the Foreground Colors

Return Value	**Method Name and Argument List**
`void`	`setStroke( Paint color )` sets the current outline color to *color*
`void`	`setFill( Paint color )` sets the current fill color to *color*

Description

The *Color* class, which is in the *javafx.scene.paint* package, defines colors using the RGB (red, green, blue) system. Any RGB color is composed of red, green, and blue components. Each component's value can range from 0 to 255 (00 to FF in hexadecimal); the higher the value, the higher the concentration of that component in the color. For example, a color with red 5 255, green 5 0, and blue 5 0 is pure red, and a color with red 5 0, green 5 0, and blue 5 255 is pure blue.

The color white is (255, 255, 255): its red, green, and blue components have the maximum value. Black is (0, 0, 0): its red, green, and blue components have the minimum value. Gray consists of equal amounts of each component. The higher the value of the components, the lighter the color of gray. This makes sense because the closer a color gets to white, the lighter that color will be. Similarly, the closer the gray value gets to 0, the darker the color of gray, because the color is approaching black.

For convenience in setting stroke and fill colors, the *Color* class provides approximately 150 *static Color* constants. **Table 4.3** lists some *Color* constants for common colors and their corresponding red, green, and blue components in hexadecimal. For a complete list of *Color* constants, see the documentation for the *Color* class in the Java Class Library.

Selected *Color* Constants and Their Red, Green, and Blue Components

Color Constant	Red	Green	Blue
■ Color.BLACK	00	00	00
■ Color.BLUE	00	00	FF
■ Color.CYAN	00	FF	FF
■ Color.DARKGRAY	A9	A9	A9
■ Color.GRAY	80	80	80
■ Color.GREEN	00	FF	00
■ Color.LIGHTGRAY	D3	D3	D3
■ Color.MAGENTA	FF	00	FF
■ Color.ORANGE	FF	A5	00
■ Color.PINK	FF	C0	CB
■ Color.RED	FF	00	00
□ Color.WHITE	FF	FF	FF
■ Color.YELLOW	FF	FF	00

Description

Each color constant is a predefined *Color* object, so you can simply assign the constant to your *Color* object reference. *Color* constants can be used wherever a *Color* or *Paint* object is expected. For example, assuming that *gc* is a *GraphicsContext* reference, this statement sets the current stroke color to orange:

```
gc.setStroke( Color.ORANGE );
```

If none of these 150 predefined colors meets our needs, we can create a custom color using any of the 16.7 million possible combinations of the component values. The *Color* class has a number of methods for creating colors, but for our purposes we'll need only the *static* factory method shown in **Table 4.4**.

A *Color* Class *static* Factory Method

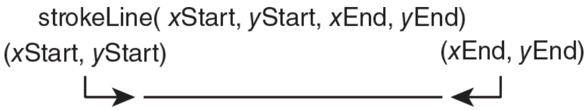
Return Value	Method Name and Argument List

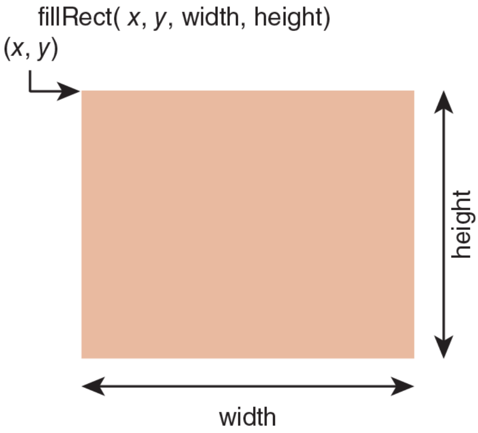
`Color`	`rgb( int rr, int gg, int bb )` creates a *Color* object with an *rr* red component, *gg* green component, and *bb* blue component

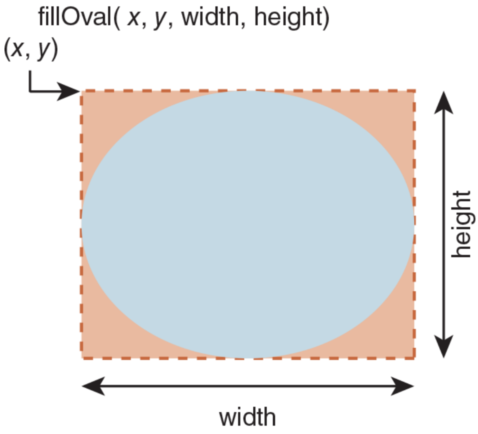
Description

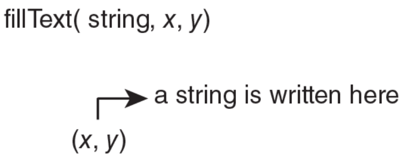
with these end-of-chapter questions

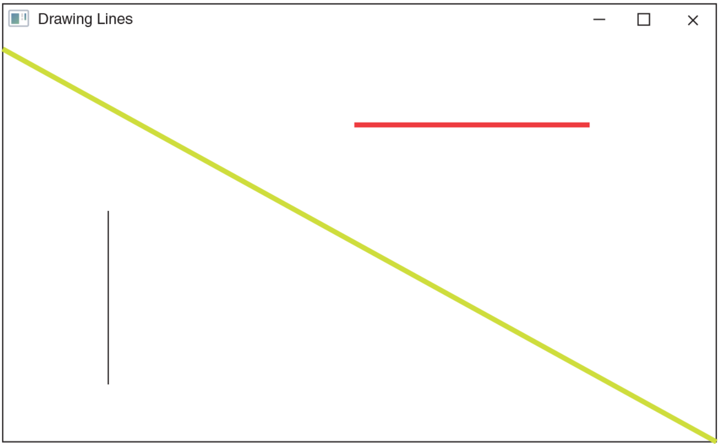
4.7.1	Multiple Choice Exercises Questions 1, 2, 3, 4, 5, 10
4.7.3	Fill In the Code Question 16
4.7.4	Identifying Errors in Code Questions 23, 24, 25, 26, 27
4.7.5	Debugging Area Questions 29, 30
4.7.8	Technical Writing Question 38

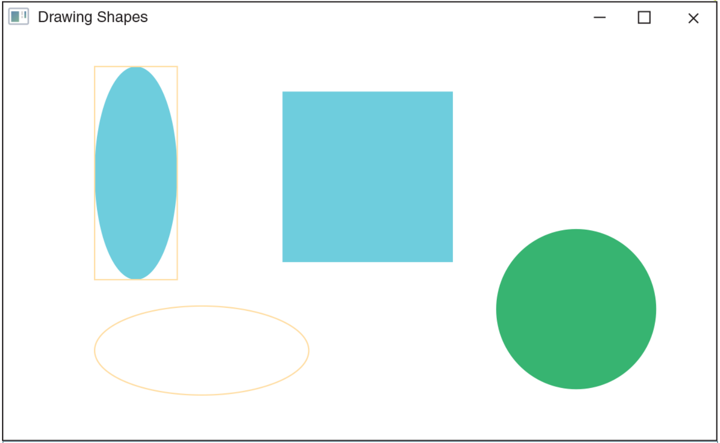

strokeLine(xStart, yStart, xEnd, yEnd)
(xStart, yStart)
(xEnd, yEnd)

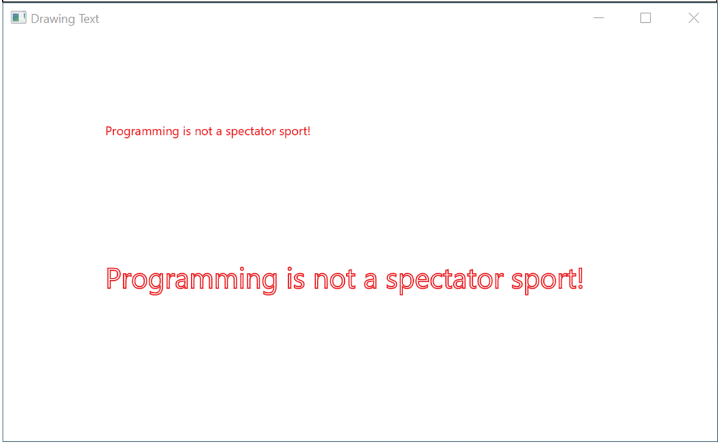

fillRect(x, y, width, height)
(x, y)
height
width


fillOval(x, y, width, height)
(x, y)
height
width


fillText(string, x, y)
a string is written here
(x, y)

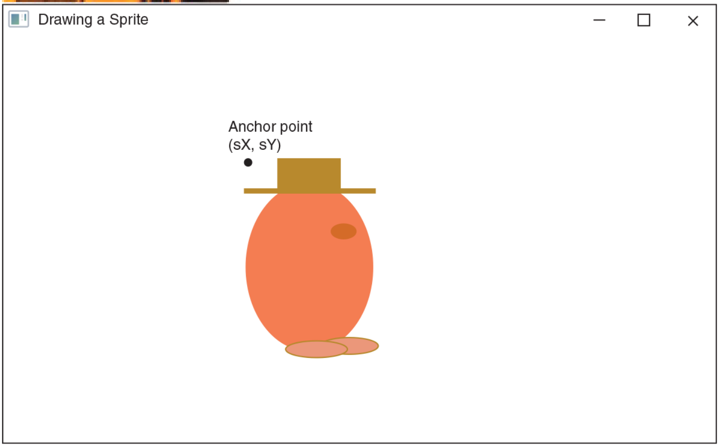

Drawing Lines


Drawing Shapes


Drawing Text

Programming is not a spectator sport!

Programming is not a spectator sport!


Drawing a Sprite
Anchor point
(sX, sY)

4.3 Drawing Shapes and Text

Now we're ready to create some graphics. Let's start with drawing lines.

Table 4.5 shows some useful methods of the *GraphicsContext* class for drawing shapes and displaying text in a window.

Drawing Methods of the *GraphicsContext* Class

Return Value	**Method Name and Argument List**
`void`	`strokeLine( double xStart, double yStart, double xEnd, double yEnd )` draws a line starting at (*xStart*, *yStart*) and ending at (*xEnd*, *yEnd*)
`void`	`strokeRect( double x, double y, double width, double height )` draws the outline of a rectangle with its top-left corner at (*x*, *y*), with the specified *width* and *height* in pixels
`void`	`fillRect( double x, double y, double width, double height )` draws a solid rectangle with its top-left corner at (*x*, *y*), with the specified *width* and *height* in pixels
`void`	`clearRect( double x, double y, double width, double height )` draws a solid rectangle in the current background color with its top-left corner at (*x*, *y*), with the specified *width* and *height* in pixels
`void`	`strokeOval( double x, double y, double width, double height )` draws the outline of an oval inside an invisible bounding rectangle with the specified *width* and *height* in pixels; the top-left

	corner of the bounding rectangle is (*x, y*)
`void`	`fillOval( double x, double y, double width, double height )` draws a solid oval inside an invisible bounding rectangle with the specified *width* and *height* in pixels; the top-left corner of the bounding rectangle is (*x, y*)
`void`	`strokeText( String s, double x, double y )` displays the *String s* with the (*x, y*) coordinate indicating the lower-left corner of the first character; the text is drawn in the current stroke color and line width
`void`	`fillText( String s, double x, double y )` displays the *String s* with the (*x, y*) coordinate indicating the lower-left corner of the first character; the text is drawn in the current fill color
`void`	`setLineWidth( double width )` sets the current stroke line width to *width* pixels
`void`	`setFont( Font f )` sets the font to *f* for displaying text

Description

As you can see, all these methods have a *void* return type, so they do not return a value. Method calls to these methods should be stand-alone statements.

The pattern for the method names is simple. The *draw* methods render the outline of the figure, while the *fill* methods render solid figures. The *clearRect* method draws a rectangle in the background color, which effectively erases anything drawn within that rectangle.

Figure 4.4 shows the relationship among the method arguments and the figures drawn.

Figure 4.4 The Arguments for Drawing Lines, Rectangles, Ovals, and Text

Description

Example 4.3 shows a graphics application that draws some lines, and **Figure 4.5** shows the output from this application. For this application, we have modified *ShellGraphicsUtility.java*, renamed it to *DrawingLines.java*, and placed our application and *JIGraphicsUtility.java* in the same folder.

Example 4.3 *DrawingLines.java*

```
/* Drawing Lines
   Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

import java.util.Random;

public class DrawingLines extends Application
{
    @Override
    public void start( Stage stage )
    {
        final int WIDTH = 700;
        final int HEIGHT = 400;
        // set up window title and size
        GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
                    stage, "Drawing Lines", WIDTH, HEIGHT );

        // draw a vertical black line
        gc.strokeLine( 100, 125, 100, 300 );

        // draw a horizontal red line
        gc.setStroke( Color.RED );
        gc.setLineWidth( 5 ); // widen the line
        gc.strokeLine( 300, 75, 550, 75 );

        // draw a diagonal line in a random color
        Random rand =  new Random( );
        int red = rand.nextInt( 0xFF );
        int green = rand.nextInt( 0xFF );
        int blue = rand.nextInt( 0xFF );
        Color randomColor = Color.rgb( red, green, blue );
        gc.setStroke( randomColor );

        gc.strokeLine( 0, 0, WIDTH - 1, HEIGHT - 1 );
    }

    public static void main( String [ ] args )
    {
        launch( args );
    }
}
```

Description

Figure 4.5 Output from *DrawingLines.java*

Description

Lines 5–8 import the JavaFX classes we use in this application, including the *Color* class. We also import the *Random* class from the standard Java Class Library on line 10.

In the call to the *setUpGraphics* method (lines 19–21), we specify the window title to be “Drawing Lines” and set the size of the window to be 700 by 400 pixels.

On lines 23–24, we draw a line by calling the *strokeLine* method, sending as arguments the start and end coordinates of the line. The line is horizontal because the *y* values for the start and end coordinates are the same. The line’s color is the default stroke color, black.

Next we draw a vertical red line (lines 26–29). We first set the stroke color to red, using the predefined *Color* constant, *Color.RED*. We make the line 5 pixels wide by calling the *setLineWidth* method with an argument of 5. This line width will stay in effect until we set the line width to a different value. The line is vertical because the *x* values for the start and end coordinates are the same.

Finally, we generate a random value for the stroke color (lines 31–37) by generating random values for the red, green, and blue components, then we use those values to create a *Color* object using the *rgb* factory method. We send FF to the *nextInt* method of the *Random* class because each red, green, and blue component’s value can range from 00 to FF in hexadecimal. Note that preceding an integer literal value with 0x indicates that the value is hexadecimal. Line 39 draws a diagonal line in this random color from the top-left corner of the window to the bottom-right corner, using the width and height of the window defined in lines 17 and 18.

Example 4.4 shows how to use the *GraphicsContext* methods for drawing shapes, and **Figure 4.6** shows the output of this application. To draw a rectangle, we call the *strokeRect* or *fillRect* methods with the (*x*, *y*) coordinate of the upper-left corner, as well as the width in pixels and the height in pixels. As you might expect, to draw a square, we specify equal values for the width and height. Line 23 draws an outlined rectangle with an upper-left corner at the coordinate (100, 50) and is 80 pixels wide and 200 pixels high; line 24 draws a solid square with an upper-left corner at the coordinate (275, 75) with sides that are 160 pixels in length.

Example 4.4 *DrawingShapes.java*

```java
/* Drawing Shapes
   Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class DrawingShapes extends Application
{
    @Override
    public void start( Stage stage )
    {
        // set up window title and size
        GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
                  stage, "Drawing Shapes", 700, 400 );
        // specify colors
        gc.setFill( Color.CYAN );
        gc.setStroke( Color.ORANGE );

        // draw rectangles
        gc.strokeRect( 100, 50, 80, 200 );  // outlined rectangle
        gc.fillRect( 275, 75, 160, 160 );   // solid square

        // draw ovals
        gc.fillOval( 100, 50, 80, 200 );    // oval inside rectangle
        gc.strokeOval( 100, 275, 200, 80 );  // outlined oval

        // draw circle using center point and radius
        gc.setFill( Color.MEDIUMSEAGREEN );
        int centerX = 550, centerY = 275;
        int radius = 75;
        gc.fillOval( centerX - radius, centerY - radius,
                     radius * 2, radius * 2 );
    }

    public static void main( String [ ] args )
    {
        launch( args );
    }
}
```

Description

Figure 4.6 Output from *DrawingShapes.java*

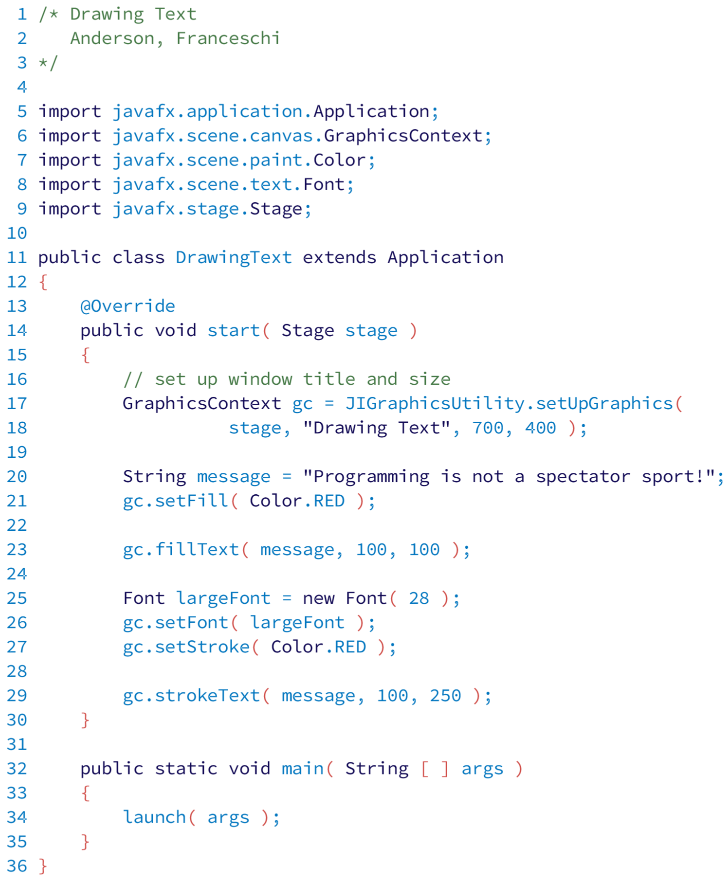
Description

Drawing an oval or a circle is a little more complex. As you can see in Figure 4.4, we need to imagine a rectangle bounding all sides of the oval or circle. Then the (*x*, *y*) coordinate we specify in the *strokeOval* or *fillOval* method is the location of the upper-left corner of the bounding rectangle. Accordingly, the width and height are the width and height of the bounding rectangle. Line 27 in Example 4.4 draws a filled oval whose upper-left corner is at coordinate (100, 50) and is 80 pixels wide and 200 pixels high. These are the same arguments we sent to the *strokeRect* method in line 23; thus, that rectangle at line 23 is the bounding rectangle for this oval, and we see that the filled oval is drawn exactly inside that rectangle. Line 28 draws an outlined oval 200 pixels wide and 80 pixels high, the same dimensions as the oval drawn at line 27 but rotated 90 degrees.

We draw a circle by calling the *strokeOval* or *fillOval* methods, specifying equal values for the width and height. Sometimes it seems more natural to identify circles by giving a center point and a radius. In this case, we can convert the center point and radius into the arguments for the *strokeOval* or *fillOval* methods as done in lines 32–35.

Example 4.5 shows how to use display text using the *fillText* method. The coordinate we specify is the lower-left corner of the first character in the *String*. The first call to *fillText* (line 23) displays our message in the default font. As you can see in **Figure 4.7**, the default text size is quite small. To display a message in a larger font, we can use the *setFont* method of the *GraphicsContext* class as shown in **Table 4.5**. First, however, we need to create a *Font* (line 25) using the *Font* constructor shown in Table 4.6. Its only argument is a requested point size for the font; in this case, we request a font size of 28. We then set the font in line 26. Line 29 redisplays the message in the larger font with a call to *strokeText* so that the text is outlined.

Example 4.5 *DrawingText.java*



```
/* Drawing Text
   Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class DrawingText extends Application
{
    @Override
    public void start( Stage stage )
    {
        // set up window title and size
        GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
                  stage, "Drawing Text", 700, 400 );

        String message = "Programming is not a spectator sport!";
        gc.setFill( Color.RED );

        gc.fillText( message, 100, 100 );

        Font largeFont = new Font( 28 );
        gc.setFont( largeFont );
        gc.setStroke( Color.RED );

        gc.strokeText( message, 100, 250 );
    }

    public static void main( String [ ] args )
    {
        launch( args );
    }
}
```

Description

Figure 4.7 Output from *DrawingText.java*

Description

A *Font* Class Constructor

Font Constructor

`Font( double size )`

creates a *Font* object with a height of *size* in the default typeface

Description

What happens if the (*x*, *y*) coordinate you specify for a figure isn't inside the window? If a figure's coordinates are outside the bounds of the window, no error will be generated, but the portion of the figure outside the window bounds won't be visible.

COMMON ERROR TRAP

If you want the whole drawing to be visible, be careful to ensure that all parts of your shape are enclosed within the bounds of the window.

CODE IN ACTION

Within the online resources, you will find a movie illustrating step-by-step how to use the *GraphicsContext* drawing methods. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

Putting all this together, we can now write an application that draws a sprite, which is a graphics object that can be positioned and drawn as a unit. **Example 4.6** shows the code to do that, and **Figure 4.8** shows our sprite.

Example 4.6 *DrawingASprite.java*

```
/* Drawing a Sprite
   Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class DrawingASprite extends Application
{
    @Override
    public void start( Stage stage )
    {
        // set up window title and size
        GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
                  stage, "Drawing a Sprite", 700, 400 );

       // drawing the sprite
       int sX = 275;
       int sY = 100;
       gc.setFill( Color.CORAL ); // body
       gc.fillOval( sX, sY + 20, 120, 160 );
       gc.setFill( Color.DARKGOLDENROD ); // hat
       gc.fillRect( sX + 30, sY, 60, 30 );
       gc.setStroke( Color.DARKGOLDENROD ); // hat brim
       gc.setLineWidth( 3 );
       gc.strokeLine( sX, sY + 30, sX + 120, sY + 30 );
       gc.setFill( Color.CHOCOLATE ); // eye
       gc.fillOval( sX + 80, sY + 60, 24, 16 );
       gc.setFill( Color.DARKSALMON ); // feet
       gc.setLineWidth( 1 );
       gc.fillOval( sX + 64, sY + 166, 60, 16 );
       gc.strokeOval( sX + 64, sY + 166, 60, 16 );
       gc.fillOval( sX + 36, sY + 170, 60, 16 );
       gc.strokeOval( sX + 36, sY + 170, 60, 16 );
    }

    public static void main( String [ ] args )
    {
        launch( args );
    }
}
```

Description

Figure 4.8 Output from *DrawingASprite.java with the Anchor Point Indicated*

Description

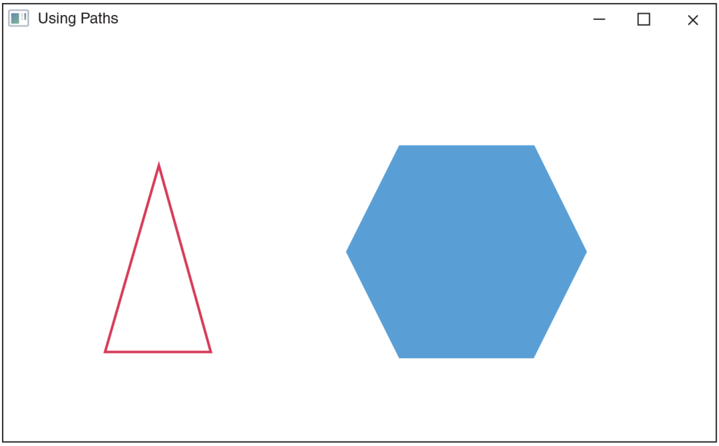
To draw our sprite, we use ovals for the body, eye, and feet, and we use a rectangle and line for the hat. We use various line widths to draw the hat brim, and we outline ovals to help define the feet.

It's important to realize that the rendering of the shapes occurs in the order in which the *stroke* or *fill* methods are executed. Any new shape that occupies the same space as a previously drawn shape will overwrite the previously drawn shape. In this drawing, we intentionally draw the eye and hat after drawing the body; and for the feet, we draw the filled ovals first, then outline the feet using stroked ovals.

In lines 21–22, we declare and initialize two variables, *sX* and *sY*. These define an **anchor point** for drawing the sprite. As indicated in Figure 4.8, the anchor point is the top-left point of the sprite's bounding rectangle. When we send *x* and *y* arguments to the *fillRect*, *strokeLine*, *fillOval*, and *strokeOval* methods, we specify those values relative to this starting (*sX*, *sY*) coordinate, or anchor point. By specifying these values relative to the anchor point—for example, *sX* 1 60—we are using **offsets**. By using offsets, we can easily change the position of the sprite on the window by simply changing the values of *sX* and *sY*. We don't need to change any of the arguments sent to the *GraphicsContext* methods. To demonstrate this, try changing the values of *sX* and *sY* and rerunning the application.

SOFTWARE ENGINEERING TIP

When drawing a figure using graphics, specify coordinates as offsets from a starting (*x*, *y*) coordinate.



4.4 Drawing Custom Shapes

To draw custom shapes we create a path, which consists of a series of coordinates between which the JavaFX platform should draw lines. Our resulting shape can be displayed either outlined or filled. Only one path can be defined at a time. **Table 4.7** shows the methods we use to create and display a path.

Path Drawing Methods of the *GraphicsContext* Class

Return Value	Method Name and Argument List
`void`	`beginPath( )` sets the drawing path to empty
`void`	`moveTo( double x, double y )` moves to the (*x*, *y*) location without adding a line to the path
`void`	`lineTo( double x, double y )` adds a line to the current path from the previous location to (*x*, *y*)

`void`	`closePath( )` closes the path by adding a line from the previous location to the first location of the path
`void`	`stroke( )` draws the path using the current stroke color
`void`	`fill( )` draws the path using the current fill color

Description

In **Example 4.7**, we illustrate drawing paths by creating an outlined triangle and a filled hexagon.

Example 4.7 *DrawingPaths.java*

```
/* Drawing Shapes Using Paths
   Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class DrawingPaths extends Application
{
    @Override
    public void start( Stage stage )
    {
        // set up window title and size
        GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
                  stage, "Using Paths", 700, 400 );

        // drawing a triangle
        // 1. create the path
        int startX = 175;
        int startY = 120;
        gc.beginPath( );
        gc.moveTo( startX, startY );
        gc.lineTo( startX - 50, startY + 175 );
        gc.lineTo( startX + 50, startY + 175 );
        gc.closePath( );

        // 2 draw the path
        gc.setStroke( Color.CRIMSON );
        gc.stroke( );

        // draw a hexagon
        // 1. create the path
        startX = 350;
        startY = 200;
        gc.beginPath( );
        gc.moveTo( startX, startY );
        gc.lineTo( startX + 50, startY - 100 );
        gc.lineTo( startX + 175, startY - 100 );
        gc.lineTo( startX + 225, startY );
        gc.lineTo( startX + 175, startY + 100 );
        gc.lineTo( startX + 50, startY + 100 );
        gc.closePath( );

        // 2. draw the path
        gc.setFill( Color.CORNFLOWERBLUE );
        gc.fill( );
    }

    public static void main( String [ ] args )
    {
        launch( args );
    }
}
```

Description

To draw our outlined triangle, we start by defining our anchor point coordinates (lines 21–22), which is the top point of the triangle. Then we call the *beginPath* method (line 23) to create an empty path. Next, we call the *moveTo* method (line 24) to set the origin of our drawing at the top point of the triangle, which is our anchor point. Of course, we could begin at any of the triangle's three points. We call the *lineTo* method (lines 25–26) twice to specify one line from the origin to the lower-left point and a second line from that location to the lower-right point.

COMMON ERROR TRAP

Forgetting to close the path could result in an incomplete shape.

We call the *closePath* method (line 27) to finish the shape; this method adds a line from our current position to the origin. Optionally, we could complete the shape by writing a third *lineTo* method call that creates a line from the lower-right point to the origin. In either case, once the path is fully defined, we draw the triangle by setting the stroke color and calling the *stroke* method (lines 29–31).

Drawing the hexagon is similar. We define coordinates for an anchor point (the leftmost point of the hexagon), begin a new path, move to our anchor point, define five lines of the hexagon using *lineTo* method calls, and close the path to finish the shape (lines 35–44). We display the filled hexagon by setting the fill color and calling the *fill* method (lines 46–49).

The resulting figures are shown in **Figure 4.9**.

Figure 4.9 Output from *DrawingPaths.java*

Description

with these end-of-chapter questions

4.7.1	Multiple Choice Exercises Questions 6, 7, 8, 9
4.7.2	Reading and Understanding Code Questions 11, 12, 13, 14, 15
4.7.3	Fill In the Code Questions 17, 18, 19, 20
4.7.4	Identifying Errors in Code

	Questions 21, 22
4.7.5	Debugging Area Question 28
4.7.6	Write a Short Program Questions 31, 32, 33
4.7.8	Technical Writing Question 39

4.5 Programming Activity 1: Writing an Application with Graphics

In this programming activity, you will create an application that uses graphics to draw a picture of your own design. The objective of this programming activity is to gain experience with the window coordinate system, the *stroke* and *fill* graphics methods, and using colors.

1. Start with the *ShellGraphicsApplication* class, change the name of the class to represent the figure you will draw, and add an *import* statement for the *Color* class. Remember to include the *JIGraphicsUtility.java* file in the same folder.
2. Create a drawing of your own design. It's helpful to sketch the drawing on graph paper first, then translate the drawing into the coordinates of the application window. Your drawing should define an anchor point, and include at least two each of rectangles, ovals, and lines, plus a path. Your drawing should also use at least three colors, one of which is a custom color.
3. Label your drawing using the *fillText* method.

Be creative and have fun with your drawing!

DISCUSSION QUESTIONS

1. If you define the starting (*x*, *y*) coordinate of the drawing as (800, 400), you might not be able to see the drawing. Explain why.
2. What is the advantage to drawing a figure using a starting (*x*, *y*) coordinate as an anchor point?

© Lois GoBe/Shutterstock.

CHAPTER REVIEW

4.6 Chapter Summary

- Graphical applications, which extend the JavaFX *Application* class, open a window and allow us to create graphical output.
- JavaFX applications use a performance/art metaphor with a stage corresponding to the window and a scene on which we can place a canvas.
- We put the code to draw shapes and text in the *start* method, which is called automatically when the application begins.
- The *GraphicsContext* class provides methods to draw on a canvas. Methods are provided for drawing figures, such as rectangles, ovals, custom shapes, and lines; to set the colors for drawing; and to display text in a window.
- An (*x*, *y*) coordinate system is used to specify locations in the window. Each coordinate corresponds to a pixel (or picture element). The *x* value specifies the horizontal position, beginning at 0 on the left-hand side of the window and increasing as you move right across the window. The *y* value specifies the vertical position, starting at 0 at the top of the window and increasing as you move down the window.
- All drawing on a graphics window is done in the current

stroke or fill foreground color, which are changed using the *setStroke* and *setFill* methods, respectively. The stroke color is used for lines and outlined shapes; the fill color is used for drawing solid shapes. Either the stroke or fill color can be used for text.

- Objects of the *Color* class can be used to set the current stroke or fill color. The *Color* class provides *static* constants for convenience in using colors.
- Custom *Color* objects can be instantiated by using the *rgb* factory method and specifying the red, green, and blue components of the color.
- When displaying text, we can set the size of the text by instantiating a *Font* object and calling the *setFont* method of the *GraphicsContext* class.
- A custom shape can be drawn using a path. We start by creating a new path and moving to our origin. We then define points to which lines should be drawn. When finished, we close the path, then either fill the path as a solid figure or stroke the path as an outlined figure.
- A sprite is a graphics object composed of shapes that is intended to move as a unit. When drawing a sprite, define a starting (x, y) coordinate as an anchor point and specify all x and y values for the drawing methods as offsets from the starting coordinate.

4.7 Exercises, Problems, and Projects

4.7.1 Multiple Choice Exercises

1. What package does the *GraphicsContext* class belong to?

    - *javafx.scene.paint*
    - *javafx.scene.canvas*
    - *javafx.stage*
    - *javafx.application*

6. If a window is 400 by 800 pixels, how many pixels does the window contain?

    - 1,200 pixels
    - 32,000 pixels
    - 320,000 pixels

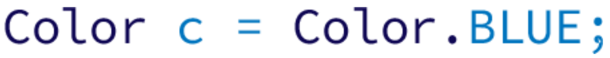
10. A JavaFX graphical application opens a window.

    - true
    - false

13. In JavaFX graphical applications, the *start* method is called automatically; the programmer does not code the method call.

    - true
    - false

16. Look at the following code:



```
Color c = Color.BLUE;
```

What is *BLUE?*

- a *static* constant of the class *Color*
- an instance variable of the class *Color*
- a *static* method of the class *Color*
- an instance method of the class *Color*

21. What can be stated about the line drawn by the following code?

```
gc.strokeLine( 100, 200, 300, 200 );
```

Description

- The line is vertical.
- The line is horizontal.
- The line is diagonal.
- None of the above.

26. What do the arguments 10, 20 represent in the following statement?

```
gc.strokeRect( 10, 20, 100, 200 );
```

Description

- the (*x*, *y*) coordinate of the upper-left corner of the rectangle we are drawing
- the width and height of the rectangle we are drawing
- the (*x*, *y*) coordinate of the center of the rectangle we are drawing
- the (*x*, *y*) coordinate of the lower-right corner of the rectangle we are drawing

31. What do the arguments 100, 200 represent in the following statement?

```
gc.strokeRect( 10, 20, 100, 200 );
```

Description

- the (*x*, *y*) coordinate of the upper-left corner of the rectangle we are drawing
- the width and height of the rectangle we are drawing
- the height and width of the rectangle we are drawing
- the (*x*, *y*) coordinate of the lower-right corner of the rectangle we are drawing

36. How many arguments does the *fillOval* method take?

    - 0
    - 2
    - 4
    - 5

41. In RGB format, a gray color can be coded as *A A A,* where the first *A* represents the amount of red in the color, the second *A* the amount of green, and the third *A* the amount of blue. *A* can vary from 0 to 255, including both 0 and 255; how many possible gray colors can we have?

    - 1
    - 2
    - 255
    - 256
    - 257

4.7.2 Reading and Understanding Code

In questions 11 through 15, assume that *gc* is a *GraphicsContext* reference.

1. In what color will the rectangle be drawn?

```
gc.setFill( Color.BLUE );
gc.fillRect( 10, 20, 100, 200 );
```

Description

2. What is the length of the line being drawn?

```
gc.strokeLine( 50, 20, 50, 350 );
```

Description

3. What is the width of the rectangle being drawn?

```
gc.fillRect( 10, 20, 250, 350 );
```

Description

4. What is the (*x*, *y*) coordinate of the upper-right corner of the rectangle being drawn?

```
gc.fillRect( 10, 20, 250, 350 );
```

Description

5. What is the (*x*, *y*) coordinate of the lower-right corner of the rectangle being drawn?

```
gc.strokeRect( 10, 20, 250, 350 );
```

Description

4.7.3 Fill In the Code

1. This code sets the current fill color to red and the current stroke color to green.

```
// assume you have a GraphicsContext object named gc
// your code goes here
```

2. This code draws the *String* “Fill In the Code” with the lower-left corner of the first character (the *F*) being at the coordinate (100, 250).

```
// assume you have a GraphicsContext object named gc
// your code goes here
```

3. This code draws a filled rectangle with a width of 100 pixels and a height of 300 pixels, starting at the coordinate (50, 30).

```
// assume you have a GraphicsContext object called gc
// your code goes here
```

4. This code draws a filled rectangle starting at (50, 30) for its upper-left corner with a lower-right corner at (100, 300).

```
// assume you have a GraphicsContext object called gc
// your code goes here
```

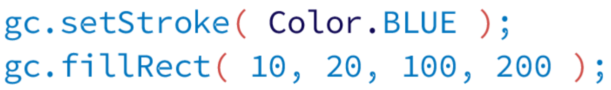
5. This code draws a circle of radius 100 with its center located at (200, 200).

```
// assume you have a GraphicsContext object called gc
// your code goes here
```

4.7.4 Identifying Errors in Code

In questions 21 through 25, assume that *gc* is a *GraphicsContext* reference.

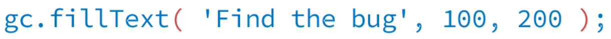
1. Where is the logic error in this code sequence that attempts to draw a solid blue rectangle?



```
gc.setStroke( Color.BLUE );
gc.fillRect( 10, 20, 100, 200 );
```

Description

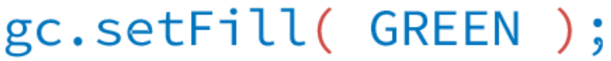
2. Where is the error in this code sequence?



```
gc.fillText( 'Find the bug', 100, 200 );
```

Description

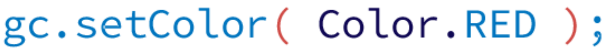
3. Where is the error in this code sequence?



```
gc.setFill( GREEN );
```

Description

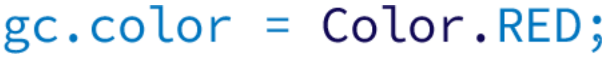
4. Where is the error in this code sequence?



```
gc.setColor( Color.RED );
```

Description

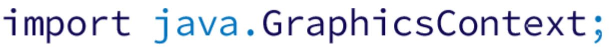
5. Where is the error in this code sequence?



```
gc.color = Color.RED;
```

Description

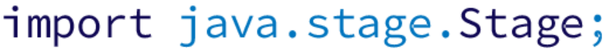
6. Where is the error in this statement?



```
import java.GraphicsContext;
```

Description

7. Where is the error in this statement?



```
import java.stage.Stage;
```

Description

4.7.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. You coded the following program in the file *MyDrawingApp.java*.

```
import javafx.application.Application;
import javafx.scene.canvas.GraphicsContext;
import javafx.stage.Stage;
public class MyDrawingApp extends Application
{
    @Override
    public void start( Stage stage )
    {
        GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
                   stage, "Tester", 700, 400 );
        gc.strokeRect( 725, 200, 50, 50 );
    }

    public static void main( String [ ] args )
    {
       launch( args );
    }
}
```

Description

You get no compiler errors, but when you run the program, the rectangle doesn't appear. Explain what the problem is and how to fix it.

2. You imported the *Color* class and coded the following on line 10 of the class *MyApplication.java*:

```
Color c = Color.rgb( 1.4, 234, 23 );    // line 10
```

Description

When you compile, you get the following message:

```
MyApplication.java:10: error: incompatible types: possible
lossy conversion from double to int
Color c = Color.rgb( 1.4, 234, 23 );    // line 10
                     ^
1 error
```

Description

Explain what the problem is and how to fix it.

3. You coded the following on line 10 of the class *MyApplication.java*:

```
Color c = Color.Blue;    // line 10
```

Description

When you compile, you get the following message:

```
MyAppliction.java:10: error: cannot find symbol
     Color c = Color.Blue;    // line 10
                    ^
symbol : variable Blue
location : class Color
1 error
```

Description

Explain what the problem is and how to fix it.

4.7.6 Write a Short Program

1. Write an application that displays the five Olympic rings.
2. Write an application that displays a tic-tac-toe board. Include a few Xs and Os.
3. Write an application that displays a rhombus (i.e., a parallelogram with equal sides). Your rhombus should not be a square.

4.7.7 Programming Projects

1. Write an application that displays two eyes. An eye can be drawn using an oval, a filled circle, and lines. On the window, write a word or two about these eyes.
2. Write an application that displays the following coins: a quarter, a dime, and a nickel. These three coins should be drawn as basic circles (of different diameters) with the currency value displayed inside the coin (for example, “25¢”). (Hint: Use the Unicode currency symbols chart to find the encoding for a cent sign.)

3. Write an application that displays a basic house made up of lines (and possibly rectangles). Your house should have multiple colors. Display a title for the house (for instance, “Java House”).

4. Write an application that displays four concentric circles. Each circle should have a lighter shade of the same color. (Hint: Look up the *brighter* method in the *Color* class.)

4.7.8 Technical Writing

1. Explain some advantages to displaying data visually as charts. Also, explain the role that color can play in communicating data values.
2. If the *strokeRect* method did not exist, but you still had the *strokeLine* method available, explain how you would be able to draw a rectangle.

4.7.9 Group Project (for a group of 1, 2, or 3 students)

1. Write an application that displays the following:
   - a drawing of two or three chessboard pieces as sprites; use a path for at least part of the drawing

- a description of one of the chessboard pieces (for instance, a rook) and its main legal moves

In order to make the description visually appealing, use several colors and several fonts.

© Lois GoBe/Shutterstock.

CHAPTER 5

Flow of Control, Part 1: Selection

CHAPTER CONTENTS

Introduction
5.1 Forming Conditions
    5.1.1 Equality Operators
    5.1.2 Relational Operators
    5.1.3 Logical Operators
5.2 Simple Selection with *if*
5.3 Selection Using *if/else*
5.4 Selection Using *if/else if*
5.5 Sequential and Nested *if/else* Statements
    5.5.1 Sequential *if/else* Statements
    5.5.2 Nested *if/else* Statements
5.6 Testing Techniques for *if/else* Statements
5.7 **Programming Activity 1: Working with *if/else***
5.8 Comparing Floating-Point Numbers
5.9 Comparing Objects

- 5.9.1 The *equals* Method
- 5.9.2 *String* Comparison Methods

5.10 The Conditional Operator (?:)
5.11 The *switch* Statement
5.12 **Programming Activity 2: Using the *switch* Statement**
5.13 Chapter Summary
5.14 Exercises, Problems, and Projects

- 5.14.1 Multiple Choice Exercises
- 5.14.2 Reading and Understanding Code
- 5.14.3 Fill In the Code
- 5.14.4 Identifying Errors in Code
- 5.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
- 5.14.6 Write a Short Program
- 5.14.7 Programming Projects
- 5.14.8 Technical Writing
- 5.14.9 Group Project

Introduction

The order of a program's instructions, called the **flow of control** of the program, is critical to producing correct results. There are essentially four types of flow of control: sequential execution, method calls, selection, and looping. Most programs use a combination of all types of flow of control.

With sequential execution in an application, the JVM executes each instruction in order. Whenever one of the instructions includes a method call, control transfers to the method, where its instructions are executed. When the method completes, we resume execution of the original instructions in order.

Sometimes, however, we don't want to execute every instruction. Some instructions should be executed only for certain input values, but not for others. For example, we may want to count only the odd numbers or perform only the operation that the user selects from a menu. For these applications, we need a way to determine at run time the input values we have and, therefore, which instructions we should execute.

In this chapter, we'll discuss **selection**, which gives us a way to test for certain conditions and to select the instructions to execute based on the results of the test. To perform selection, Java provides a number of alternatives: *if*, *if/else*, *if/else if*, the conditional operator (?:), and *switch*.

```
75 > 18 && 75 < 65 is false
75 < 65 && 75 > 18 is false
75 > 65 || 75 < 18 is true
75 > 65 || 75 < 18 && false is true
( 75 > 65 || 75 < 18 ) && false is false
```

5.1 Forming Conditions

Often in a program, we need to compare variables or objects. For instance, to determine whether someone can shop online, we want to know if the person's age is at least 18. Or, if we are adding students to the honor roll, we want to add only those students with averages of 90 or better. Similarly, if we are sending warnings to students, we want to send those warnings only to students whose averages are below 60.

Java provides equality, relational, and logical operators to evaluate and test whether an expression is true or false. It also provides selection statements to transfer control to a different part of the program depending on the result of that test.

5.1.1 Equality Operators

A common operation is to compare two variables or values of the same data type to determine if their values are equal. For example, we need to compare the user's input to a 'y' to determine whether he or she wants to play a game again. Or if we want to print a list of students who will continue next year, we need to eliminate the students who are graduating seniors.

To compare values of primitive data types, Java provides the **equality operators** shown in **Table 5.1**. Both are binary operators, meaning that they take two operands. The operands may be expressions that evaluate to a primitive numeric or *boolean* type or an object reference. The result of an expression composed of a relational operator and its two operands is a *boolean* value, that is, *true* or *false*.

Equality Operators

Equality Operator	**Type**	**Meaning**
==	binary	is equal to
!=	binary	is not equal to

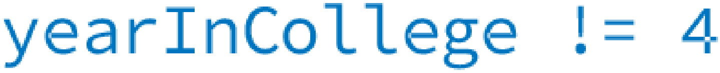
Description

For instance, if an *int* variable *age* holds the value 32, then

the expression `( age == 32 )` will evaluate to *true*, and

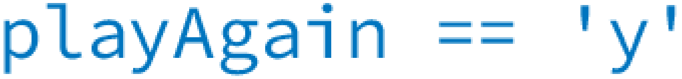
the expression `( age != 32 )` will evaluate to *false*.

The following expression can be used to eliminate seniors by testing whether the value of the *int* variable *yearInCollege* is not equal to 4:



```
yearInCollege != 4
```

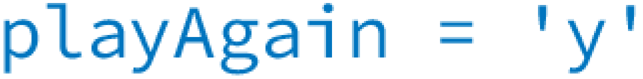
The following expression can be used in a game program to determine whether the user wants to play again:



```
playAgain == 'y'
```

Assuming the user’s input is stored in the *char* variable *playAgain*, then if the user typed ‘y’, the expression evaluates to *true*; with any other input value, the expression evaluates to *false*.

A common error is to use the assignment operator instead of the equality operator. For example:



```
playAgain = 'y'
```

actually **assigns** the value *y* to the variable *playAgain*.

COMMON ERROR TRAP

Do not confuse the equality operator == (double equal signs) with the assignment operator = (one equal sign).

Although the equality operators can be used to compare object references, these operators cannot be used to compare object data. We discuss the comparison of objects later in the chapter.

5.1.2 Relational Operators

To compare values of primitive numeric types, Java provides the **relational operators** shown in **Table 5.2**. These operators are binary, meaning that they take two operands, each of which is an expression that evaluates to a primitive numeric type. The relational operators cannot be used with *boolean* expressions or with object references.

Relational Operators

Relational Operator	Type	Meaning
<	binary	is less than
<=	binary	is less than or equal to
>	binary	is greater than
>=	binary	is greater than or equal to

Description

Again, if an *int* variable *age* holds the value 32, then

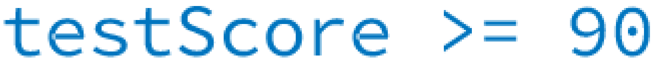
the expression `( age < 32 )` will evaluate to *false*,

the expression `( age <= 32 )` will evaluate to *true*,

the expression `( age > 32 )` will evaluate to *false*, and

the expression `( age >= 32 )` will evaluate to *true*.

This expression tests whether an *int* variable *testScore* is at least 90:



```
testScore >= 90
```

Description

This code tests whether that test score is less than 60:

```
testScore < 60
```

5.1.3 Logical Operators

A common operation in a program is to test whether a combination of conditions is true or false. For these operations, Java provides the **logical operators** !, &&, and ||, which correspond to the Boolean logic operators NOT, AND, and OR. These operators, which are shown in **Table 5.3**, take *boolean* expressions as operands. A *boolean* expression can be a combination of variables, operators, and method calls that result in a *boolean* value.

Logical Operators

Logical Operator	Type	Meaning
`!`	unary	NOT
`&&`	binary	AND
`||`	binary	OR

Description

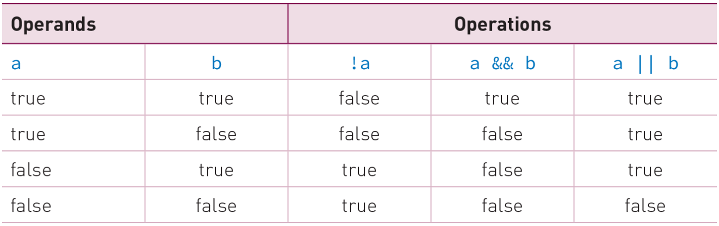
The NOT operator (!) takes one *boolean* expression as an operand and inverts the value of that operand. If the operand is *true*, the result will be *false*; and if the operand is *false*, the result will be *true*.

The AND operator (&&) takes two *boolean* expressions as operands; if both operands are *true*, then the result will be *true*; otherwise, the result will be *false*.

The OR operator (||) also takes two *boolean* expressions as operands. If both operands are *false*, then the result will be *false*; otherwise, it will be *true*. The OR operator consists of two vertical bars with no intervening space. On the PC keyboard, the vertical bar is the shifted character above the *Enter* key.

The truth table for these logical operators is shown in **Table 5.4**.

Truth Table for Logical Operators



Operands		Operations		
a	b	!a	a && b	a || b
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

Description

REFERENCE POINT

The complete Operator Precedence Chart is provided in Appendix B.

The order of precedence of the relational and logical operators is shown in **Table 5.5**, along with the arithmetic operators. Note that the Unary NOT operator (!) has the highest precedence of the relational and logical operators, followed by the relational operators, then the equality operators, then AND (&&), then OR (||).

Operator Precedence

Operator Hierarchy	Order of Same-Statement Evaluation	Operation
`( )`	left to right	parentheses for explicit grouping
`++, --`	right to left	shortcut postincrement/ postdecrement
`++, --, !`	right to left	shortcut preincrement/ predecrement, logical

		unary NOT
`*, /, %`	left to right	multiplication, division, modulus
`+, -`	left to right	addition or *String* concatenation, subtraction
`<, <=, >, >=`	left to right	relational operators: less than, less than or equal to, greater than, greater than or equal to
`==, !=`	left to right	equality operators: equal to and not equal to
`&&`	left to right	logical AND
`||`	left to right	logical OR
`=, +=, -=, *=, /=, %=`	right to left	Assignment operator and shortcut assignment operators

Description

Example 5.1 shows these operators at work.

Example 5.1 How Logical Operators Work

```
 1 /* Using Logical Operators
 2    Anderson, Franceschi
 3 */
 4
 5 public class LogicalOperators
 6 {
 7  public static void main( String [ ] args )
 8  {
 9   int age = 75;
10   boolean test;
11
12   test = ( age > 18 && age < 65 );
13   System.out.println( age + " > 18 && " + age + " < 65 is " + test );
14
15   // short circuitry with AND
16   test = ( age < 65 && age > 18 );
17   System.out.println( age + " < 65 && " + age + " > 18 is " + test );
18
19   // short circuitry with OR
20   test = ( age > 65 || age < 18 );
21   System.out.println( age + " > 65 || " + age + " < 18 is " + test );
22
23   // AND has higher precedence than OR
24   test = ( age > 65 || age < 18  && false );
25   System.out.println( age + " > 65 || " + age
26                       + " < 18 && false is " + test );
27
28   // use of parentheses to force order of execution
29   test = ( ( age > 65 || age < 18 )  && false );
30   System.out.println( "( " + age + " > 65 || " + age
31                       + " < 18 ) && false is " + test );
32  }
33 }
```

Description

Line 12 evaluates whether the variable *age* is greater than 18 and less than 65 and assigns the result to the *boolean* variable *test*. Since line 9 set the value of *age* to 75, the first operand (*age . 18*) evaluates to *true*. The second operand (*age , 65*) evaluates to *false*; finally,

```
true && false
```

evaluates to *false*, and *false* is assigned to *test*, which is printed at line 13. Line 16 evaluates the same expression as in line 12, but in reverse order. Now the first operand (*age , 65*) evaluates to *false*, and therefore, since the operator is the logical AND, the overall expression evaluates to *false*, independently of the value of the second operand. Because (*false* && something) always evaluates to *false*, the second operand (*age . 18*) will never be evaluated by the JVM. This is called **short-circuit**

evaluation.

Line 20 shows an example of short-circuit evaluation for the logical OR operator. The first operand (*age > 65*) evaluates to *true*, resulting in the overall expression evaluating to *true*, independently of the value of the second operand. Because (*true* || something) always evaluates to *true*, the second operand will never be evaluated by the JVM.

As shown in Table 5.5, the logical AND operator has higher precedence than the logical OR operator. Thus, the expression in line 24 is not evaluated from left to right; rather, the second part of the expression (*age , 18* && *false*) is evaluated first, which evaluates to *false*. Then (*age . 65* || *false*) evaluates to *true*, which is assigned to *test*, and then output at lines 25–26. If we want to evaluate the expression from left to right, we have to use parentheses to force this, as in line 29. Then, (*age > 65* || *age , 18*) is evaluated first and evaluates to *true*; (*true* && *false*) is evaluated next and evaluates to *false*.

Figure 5.1 shows the output of Example 5.1.

Figure 5.1 Output from Example 5.1

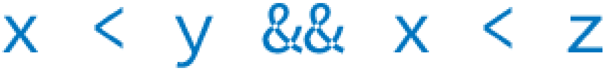
Description

Suppose we have three *ints: x, y*, and *z*, and we want to test if *x* is less than both *y* and *z*. A common error is to express the condition this way:

```
x < y && z // incorrect comparison of x to y and z
```

Description

Because *z* is not a *boolean* variable, this statement will generate a compiler error. Both operands of the logical AND and logical OR operators must evaluate to a *boolean* expression. The correct expression is the following:



```
x < y && x < z
```

Description

COMMON ERROR TRAP

Be sure that both operands of the logical AND and logical OR operators are *boolean* expressions. Expressions such as `x < y && z`, with *x, y,* and *z* being numeric types, are illegal. Instead, use the expression `x < y && x < z`

There are often several ways to express the same condition using the Java logical operators. For instance, suppose we have two boolean variables called *flag1* and *flag2*, and we want to test if at least one of them is *false*.

In plain English, we would translate it as *flag1* is *false* OR *flag2* is *false*.

Table 5.6 provides several equivalent expressions for the preceding test.

Examples of Equivalent Expressions

Equivalent Expressions	English Meaning
`( flag1 == false ) || ( flag2 == false )`	*flag*1 is false OR *flag*2 is false
`!flag1 || !flag2`	!*flag*1 is true OR !*flag*2 is true
`! ( flag1 && flag2 )`	not both *flag*1 and *flag*2 are true

Description

Although all the expressions in Table 5.6 are equivalent, the first expression, which is the simplest translation of the condition to test, is the easiest to understand and would be the best selection for readability.

DeMorgan’s Laws

Thanks to the work of the British mathematician Augustus DeMorgan, we have a set of rules to help develop expressions that are equivalent. DeMorgan, who is known for his work in Boolean algebra and set theory, developed what are known as DeMorgan’s Laws. They are the following:

1. `NOT( A AND B ) = ( NOT A ) OR ( NOT B )`
2. `NOT( A OR B ) = ( NOT A ) AND ( NOT B )`

Description

In Java, therefore, using the first law, we see that

`!( a && b )` is equivalent to `!a || !b`

Description

Using the second law, we see that

`!( a || b )` is equivalent to `!a && !b`

Description

These laws are illustrated in the extended truth table shown in **Table 5.7**.

Truth Table for DeMorgan's Laws

a	b	!a	!b	a && b	a || b	!(a && b)	!a || !b	!(a || b)	!a && !b
true	true	false	false	true	true	false	false	false	false
true	false	false	true	false	true	true	true	false	false
false	true	true	false	false	true	true	true	false	false
false	false	true	true	false	false	true	true	true	true

Description

Thus, to use DeMorgan's Laws, we change the AND operator to OR and change the OR operator to AND, and apply the NOT operator (!) to each operand of a logical operator. When the operands are expressions using relational or equality operators, the negated expressions are shown in **Table 5.8**.

The Logical NOT Operator Applied to Relational and Equality Operators

Expression	! (Expression)
a == b	a != b
a != b	a == b
a < b	a >= b
a >= b	a < b
a > b	a <= b
a <= b	a > b

Description

SOFTWARE ENGINEERING TIP

Compose Boolean expressions so that they are easy to read and understand.

For instance, suppose we have an *int* variable named *age*, representing the age of a person, and we want to assess whether *age* is less than or equal to 18 or greater than or equal to 65.

Table 5.9 provides several equivalent expressions for the preceding test.

Again, although all the expressions in Table 5.9 are equivalent, the first expression, which is the simplest translation of the condition to test, is the easiest to read.

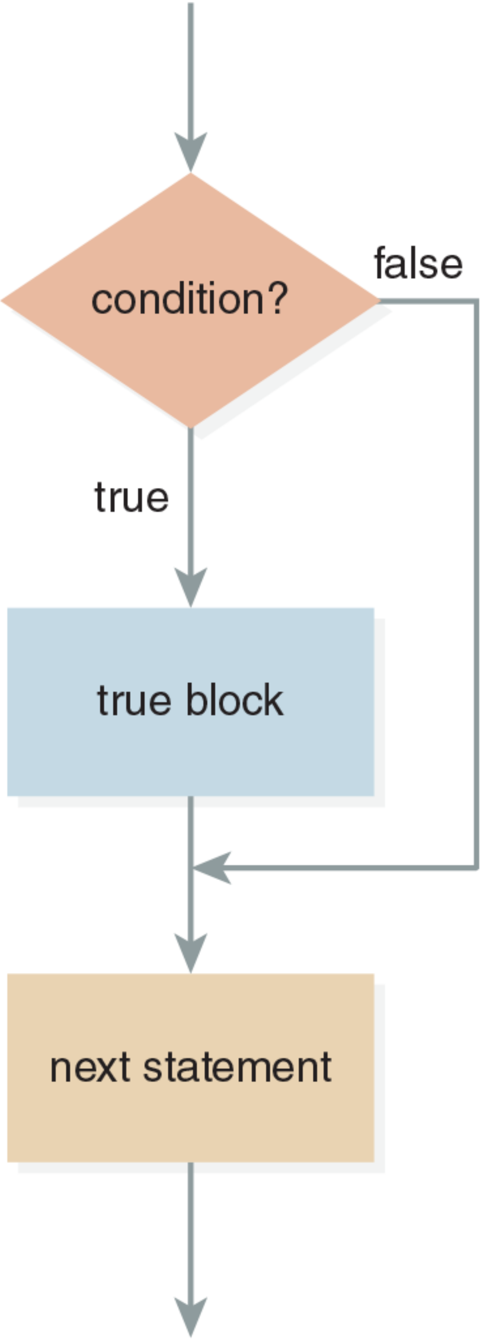
More Examples of Equivalent Expressions

Equivalent Expressions	English Meaning
`( age <= 18 || age >= 65 )`	*age* is less than or equal to 18 or *age* is greater than or equal to 65
`!( age > 18 && age < 65 )`	*age* is not between 18 and 65
`!( age > 18 ) || !( age < 65 )`	*age* is not greater than 18 or *age* is not less than 65

Description

with these end-of-chapter questions

5.14.1	Multiple Choice Exercises Questions 1, 2, 3, 4, 5, 6, 7
5.14.2	Reading and Understanding Code Questions 10, 11
5.14.4	Identifying Errors in Code Questions 31, 32

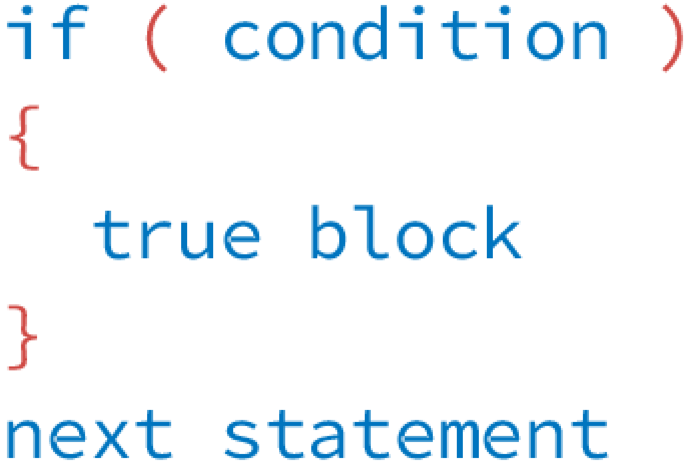

condition?
false
true
true block
next statement

```
Enter your test grade > 85
Enter your extra credit > 10
Your final test grade is 95
```

```
Enter your test grade > 85
Enter your extra credit > 0
Your final test grade is 85
```

5.2 Simple Selection with *if*

The simple selection pattern is appropriate when our program needs to perform an operation for one set of data, but not for all other data. For this situation, we use a simple *if* statement, which has this pattern:



```
if ( condition )
{
   true block
}
next statement
```

Description

The true block can contain one or more statements and is executed only if the condition evaluates to *true*. After the true block executes, the instruction following the *if* statement is executed. If the condition is *false*, the true block is skipped and execution picks up at the next instruction after the *if* statement. If the true block contains only one statement, the curly braces are optional. **Figure 5.2** illustrates the flow of control of a simple *if* statement.

Figure 5.2 Flow of Control of a Simple *if* Statement

Description

In **Example 5.2**, we first prompt the user to enter a grade at lines 12–13. Then we prompt the user for any extra credit points at lines 15–16. At line 18, we test whether the extra credit points are greater than 0. If so, we add the extra credit points to the test grade at line 19. Then, no matter what the extra credit was, lines 21–22 are executed, which print the final grade. **Figures 5.3** and **5.4** show two runs of the program, one with extra credit greater than 0, and one with no extra credit.

Example 5.2 Working with *if* Statements

```
 1 /* Using if to calculate a final test grade
 2    Anderson, Franceschi
 3 */
 4 import java.util.Scanner;
 5
 6 public class TestGrade
 7 {
 8   public static void main( String [ ] args )
 9   {
10     Scanner scan = new Scanner( System.in );
11
12     System.out.print( "Enter your test grade > " );
13     int grade = scan.nextInt( );
14
15     System.out.print( "Enter your extra credit > " );
16     int extraCredit = scan.nextInt( );
17
18     if ( extraCredit > 0 )
19         grade += extraCredit;
20
21     System.out.println( "Your final test grade is "
22                   + grade );
23   }
24 }
```

Description

Figure 5.3 Output of Example 5.2 with 10 Extra Credit Points

Description

Figure 5.4 Output of Example 5.2 with No Extra Credit

Description

Notice the indentation of the true block (line 19). Indenting clarifies the structure of the program. It's easy to see that we add the extra credit to the test grade only if the condition is true. Notice also that we skipped a line after the end of the *if* statement; this further separates the true block from the instruction that follows the *if* statement, making it easier to see the flow of control.

SOFTWARE ENGINEERING TIP

Indent the true block in an *if* statement for clarity.

Many software engineers believe it's a good practice to include the curly braces even if only one statement is included in the true block, because it increases clarity and ease of maintenance. The curly braces increase clarity because they highlight the section of code to be executed when the condition is *true*. Program maintenance is easier because if the program requirements change and we need to add a second statement to the true block, the curly braces are already in place.

Note that there is no semicolon after the condition. If we place a semicolon after the condition, as in this **incorrect** statement,

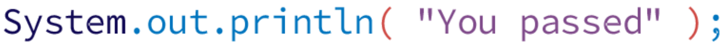
```
if ( grade >= 60 ); // incorrect to place semicolon here
      System.out.println( "You passed" );
```

Description

COMMON ERROR TRAP

Adding a semicolon after the condition of an *if* statement indicates that the true block is empty and can cause a logic error at run time.

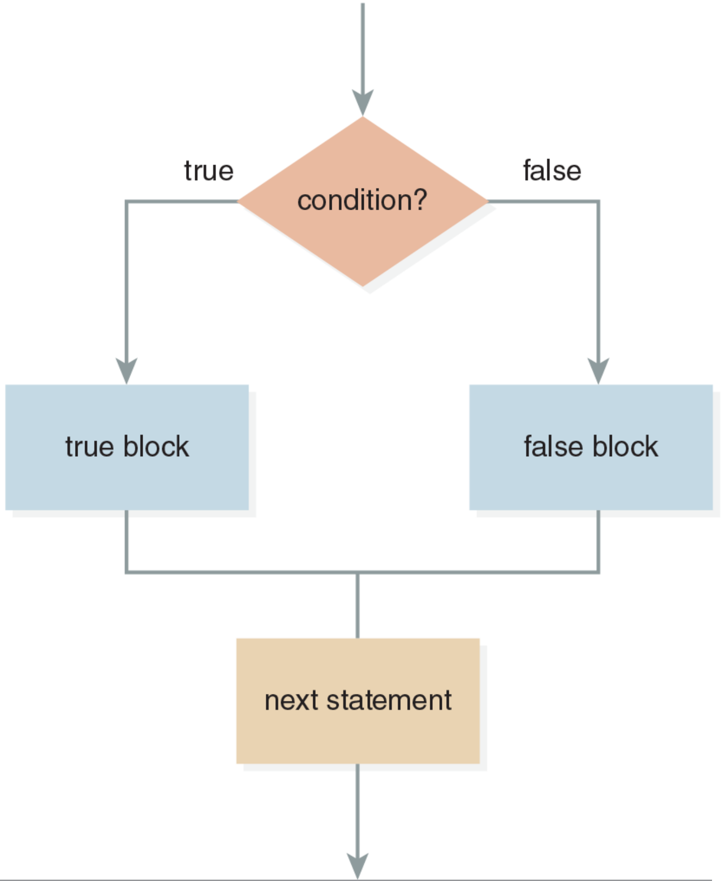
the compiler will not generate an error. Instead, it will consider the semicolon to indicate that the true block of the *if* statement is empty, because a semicolon by itself indicates a statement that does nothing. In this case, the compiler concludes that there is no instruction to execute when the condition is *true*. As a result, when the program runs, the statement



```
System.out.println( "You passed" );
```

Description

is treated as though it follows the *if* statement, and therefore, the message `“You passed”` will be printed regardless of the value of *grade*.

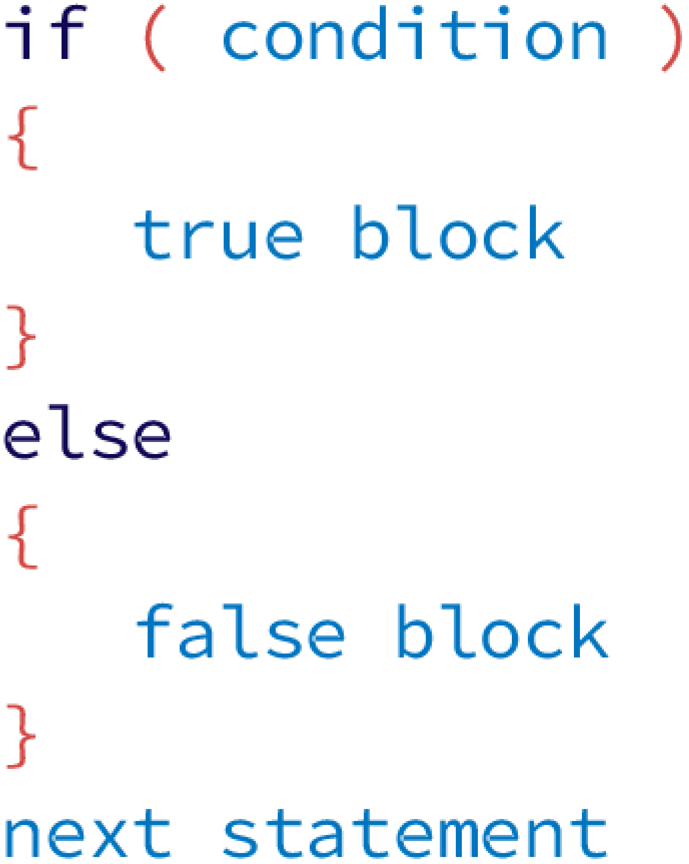

true
condition?
false
true block
false block
next statement

```
Enter a grade > 60
You passed
```

```
Enter a grade > 59
You failed
```

5.3 Selection Using *if/else*

The second form of an *if* statement is appropriate when the data falls into two mutually exclusive categories and different instructions should be executed for each category. For these situations, we use an *if/else* statement, which has the following pattern:



```
if ( condition )
{
    true block
}
else
{
    false block
}
next statement
```

Description

If the condition evaluates to *true*, the true block is executed and the false block is skipped. If the condition evaluates to *false*, the true block is skipped and the *false* block is executed. In either situation, the statement following the *if* statement is executed next. **Figure 5.5** illustrates the flow of control of an *if/else* statement.

Figure 5.5 Flow of Control of an *if/else* Statement

Description

If the true or false block contains only one statement, the curly braces are optional for that block.

Again, notice the indentation of the true and false blocks and that the *else* and curly braces line up under the *if*. This coding style makes it easy to see which statements belong to the true block and which belong to the false block. If the indentation is incorrect, a reader of our program may misunderstand which statements will be executed. In any event, the compiler ignores the indentation; the indentation is designed only to make it easier for humans to understand the logic of the code.

In **Example 5.3**, we test a grade to determine whether it is a passing grade (.5 60) or a failing grade (any other value). This is a case where the data is mutually exclusive: either the grade is a passing grade or it is not. We want to print a different message depending on the grade. After prompting the user for a numeric grade, we declare a *String* to hold the appropriate message (line 16), which will be determined in our *if/else* statement. If the *if* condition, `(grade >= 60)`, is true, then we assign “You passed” to *message*. If the condition is false, we assign “You failed” to *message*. On line 22, after the *if/else* statement completes, we print whatever *String* we have assigned to *message*. **Figures 5.6** and **5.7** show two runs of the program, first with a grade greater than or equal to 60, and then with a grade less than 60.

Example 5.3 Working with *if/else* Statements

```
/* Using if/else
   Anderson, Franceschi
*/

import java.util.Scanner;

public class PassingGrade
{
 public static void main( String [ ] args )
 {
    Scanner scan = new Scanner( System.in );

    System.out.print( "Enter a grade > " );
    int grade = scan.nextInt( );

    String message;
    if ( grade >= 60 )
     message = "You passed";
    else
     message = "You failed";

    System.out.println( message );
 }
}
```

Description

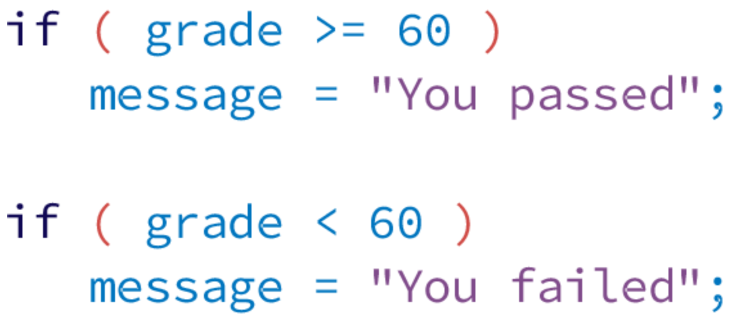
Figure 5.6 Output from Example 5.3 with *grade* >= 60

Description

Figure 5.7 Output from Example 5.3 with *grade* < 60

Description

Note that we could have used two sequential *if* statements, as in:



```
if ( grade >= 60 )
   message = "You passed";

if ( grade < 60 )
   message = "You failed";
```

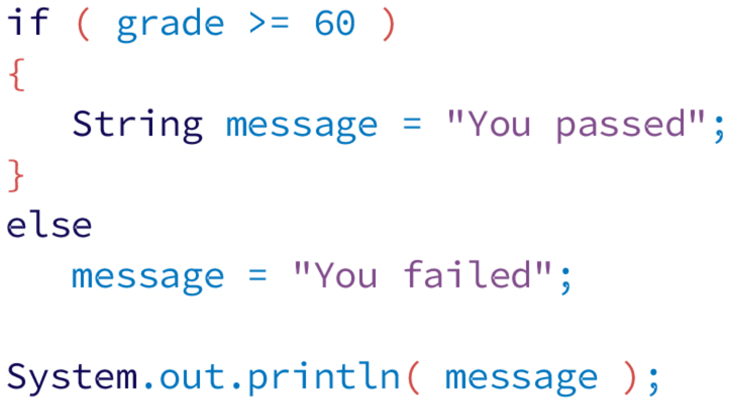
Description

However, if the first condition, `(grade >= 60)`, is false, the second condition, `(grade < 60)`, must be true. So an *if/else* simplifies our processing and avoids unnecessarily testing two conditions when only one of the conditions can be true.

Block Scope

The **scope** of a variable is the region within a program where the variable can be **referenced**, or used. When we declare a variable, its scope extends from the point at which it is declared until the end of the block in which we declared it. A method, such as *main*, is a block. Thus, in Example 5.3, the scope of the object reference *scan* extends from line 11 through the end of *main*. Thus, we can legally reference *scan* on line 14. Similarly, the scope of *grade* extends from its declaration (line 14) through the end of *main*, and we can legally reference it on line 17 in the *if* condition. Finally, the scope of the *String message* extends from line 16 through the end of *main*, and thus we can legally reference *message* on lines 18, 20, and 22.

The true blocks and false blocks for *if* statements are also blocks. Thus, if instead of declaring the *String message* on line 16, we declare it inside the true block of the *if* statement as in the following,



```
if ( grade >= 60 )
{
   String message = "You passed";
}
else
   message = "You failed";

System.out.println( message );
```

Description

then the scope of *message* extends from its declaration only until the end of the true block. In this case, the compiler will generate “cannot find symbol” error messages for the references to *message* inside the false block and for the *System.out.println* statement after the *if* statement because *message* is out of scope outside of the true block.

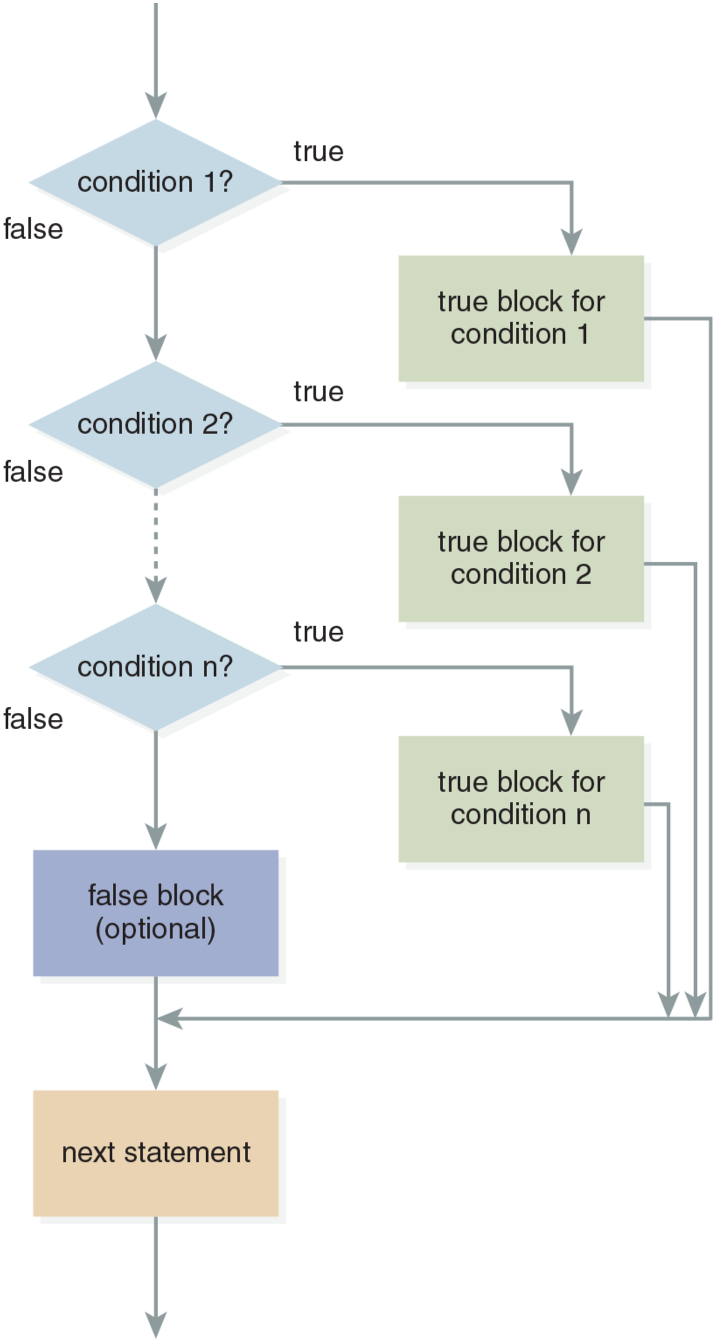
CODE IN ACTION

Within the online resources, you will find a movie illustrating step-by-step how to use an *if/else* statement. Click on this chapter’s link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

with these end-of-chapter questions

5.14.2	Reading and Understanding Code Questions 12, 13
5.14.3	Fill In the Code Questions 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30
5.14.4	Identifying Errors in Code Questions 33, 34, 35
5.14.5	Debugging Area Question 40
5.14.6	Write a Short Program Questions 42, 43, 46, 48
5.14.8	Technical Writing


condition 1?
true
false
true block for condition 1
condition 2?
true
false
true block for condition 2
condition n?
true
false
true block for condition n
false block (optional)
next statement

```
Enter your test grade: 83
Your test grade of 83 is a letter grade of B
```

5.4 Selection Using *if/else if*

The last form of an *if* statement is appropriate when the data falls into more than two mutually exclusive categories and the appropriate instructions to execute are different for each category. For this situation, Java provides the *if/else if* statement.

The *if/else if* statement follows this pattern:

```
if ( condition 1 )
{
      true block for condition 1
}
else if ( condition 2 )
{
      true block for condition 2
}
. . .
else if ( condition n )
{
      true block for condition n
}
else
{
      false block for all conditions being false
}
next statement
```

Description

The flow of control for this form of the *if* statement is shown in **Figure 5.8**.

Figure 5.8 Flow of Control of an *if/else if* Statement

Description

There can be any number of conditions in an *if/else if* statement. As we can see, once a condition evaluates to *true* for any value, control moves to the true block for that condition, then skips the remainder of the conditions, continuing execution at any statement that follows the *if/else if* statement. The final false block (along with the final *else*) is optional and is executed only when none of the conditions evaluates to *true*. Note that if the final *else* is used, then the condition it covers is not coded. The *else* stands alone on the line.

We can use the *if/else if* statement to determine a student's letter grade based on his or her numeric grade. **Example 5.4** demonstrates a Java application that prompts a student for a test grade and translates that grade into a letter grade.

Example 5.4 A Demonstration of *if/else if*

```java
/* A program to translate a numeric grade into a letter grade
   Anderson, Franceschi
*/

import java.util.Scanner;

public class LetterGrade
{
  public static void main( String [ ] args )
  {
    Scanner scan = new Scanner( System.in );

    char letterGrade;

    System.out.print( "Enter your test grade: " );
    int grade = scan.nextInt( );

    if ( grade >= 90 )
         letterGrade = 'A';

    else if ( grade >= 80 )
         letterGrade = 'B';

    else if ( grade >= 70 )
         letterGrade = 'C';

    else if ( grade >= 60 )
         letterGrade = 'D';

    else  // grade fits none of the conditions
          letterGrade = 'F';

    System.out.println( "Your test grade of " + grade
                        + " is a letter grade of " + letterGrade );
  }
}
```

Description

COMMON ERROR TRAP

In an *if/else* or *if/else if/else* statement, do not specify a condition for the final *else*.

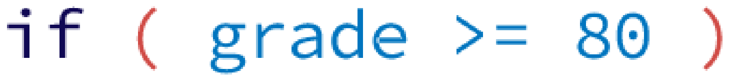
Figure 5.9 shows the output from the program when a student enters a grade of 83.

Figure 5.9 Output from Example 5.4

Description

Notice that each condition is a simple relational expression.

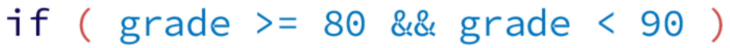
Even though we assign a *B* letter grade when the numeric grade is between 80 and 89, the condition for a *B* letter grade (line 21) is simply:



```
if ( grade >= 80 )
```

Description

We don’t need to write the condition as



```
if ( grade >= 80 && grade < 90 )
```

Description

because by the time the condition is tested at line 21, all numeric grades greater than or equal to 90 have been eliminated by the test condition at line 18. Any grade greater than or equal to 90 causes the condition at line 18, `( grade >= 90 )`, to evaluate to *true*. For those grades, the flow of control is to assign an *A* to *letterGrade* at line 19, then skip the remainder of the conditions, continuing execution at the statement following the *if/else if* statement, which is line 33 in this example. Thus, if the condition at line 21 is evaluated, we know that the *grade* must be less than 90. Note that in line 30, the final *else* covers the only other possibility: that the grade is less than 60. We do not code that condition; rather we include only the *else*.

CODE IN ACTION

Within the online resources, you will find a movie illustrating step-by-step how to use an *if/else if* statement. Click on this chapter’s link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

with these end-of-chapter questions

5.14.1	Multiple Choice Exercises Question 8
5.14.2	Reading and Understanding Code

5.14.4

Question 15
Identifying Errors in Code
Questions 36, 37, 38

```
Enter the first integer: 6
Enter the second integer: 7
Enter the third integer: 5
The smallest is 5
```

```
I'm thinking of a number between 1 and 10. What is your guess? 2
The number was 10
You missed it by a mile!
Better luck next time.
```

5.5 Sequential and Nested *if/else* Statements

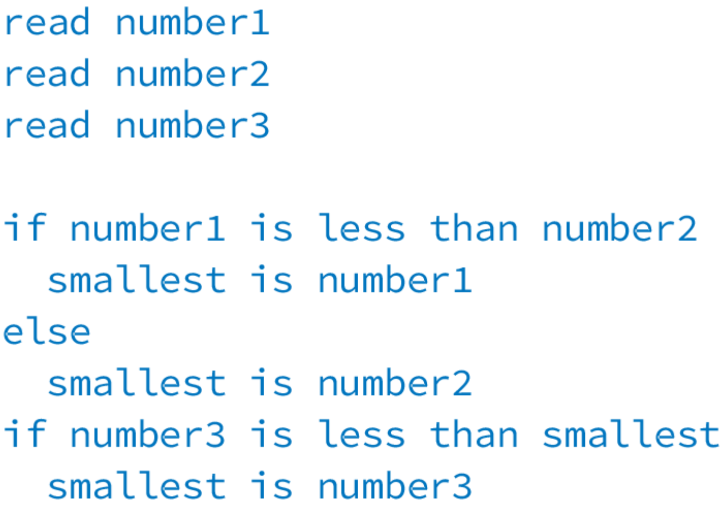
When we need the results of one *if* statement's processing before we can evaluate the next condition, we can write multiple *if* statements either sequentially or nested within other *if* statements.

5.5.1 Sequential *if/else* Statements

Finding the Minimum or Maximum Values

To illustrate sequential *if* statements, let's look at the problem of finding the smallest of three numbers.

We can use multiple, sequential *if* statements. First we find the smaller of the first two numbers, then we find the smaller of that result and the third number. The pseudocode for this application is:



```
read number1
read number2
read number3

if number1 is less than number2
  smallest is number1
else
  smallest is number2
if number3 is less than smallest
  smallest is number3
```

Description

Translating the pseudocode into Java, we get the application in **Example 5.5**, which prompts the user for three integers and outputs the smallest of the three numbers. In this application, we use two sequential *if* statements. The first *if* statement (lines 23–26) uses an *if/else* statement to find the smaller of the first two integers and stores that value into the variable *smallest*. Then, the second *if* statement (lines 28–29) compares the third integer to the value stored in *smallest*. In the second *if* statement, we don’t use an *else* clause, because we need to change the value in *smallest* only if the condition is *true*, that is, if the third number is less than *smallest*. Otherwise, the smallest value is already stored in *smallest*.

Example 5.5 An Application with Sequential *if* Statements

```
/* Find the smallest of three integers
   Anderson, Franceschi
*/

import java.util.Scanner;

public class FindSmallest
{
  public static void main( String [ ] args )
  {
       int smallest;
       int num1, num2, num3;

       Scanner scan = new Scanner( System.in );

       System.out.print( "Enter the first integer: " );
       num1 = scan.nextInt( );
       System.out.print( "Enter the second integer: " );
       num2 = scan.nextInt( );
       System.out.print( "Enter the third integer: " );
       num3 = scan.nextInt( );

       if ( num1 < num2 )
          smallest = num1;
       else
          smallest = num2;

       if ( num3 < smallest )
          smallest = num3;

       System.out.println( "The smallest is " + smallest );
  }
}
```

Description

When the program in Example 5.5 is run using 6, 7, and 5 for the three integers, the output is as shown in **Figure 5.10**.

Figure 5.10 Output from Example 5.5

Description

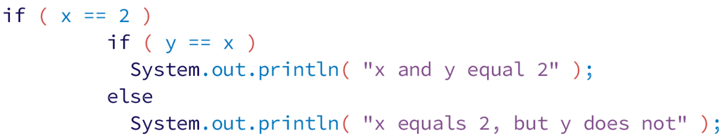
One more point. The code only checks that one number is less than another. What happens if two or more of the numbers are equal? The code still works! We only need to find the smallest value; we don’t care which of the variables holds that smallest value.

5.5.2 Nested *if/else* Statements

If statements can be written as part of the true or false block of another *if* statement. These are called nested *if* statements. Typically, we nest *if* statements when more information is required beyond the results of the first *if* statement.

One difficulty that arises with nested *if* statements is specifying which *else* clause pairs with which *if* statement, especially if some *if* statements have *else* clauses and others do not. The compiler matches any *else* clause with the most previous *if* statement that doesn't already have an *else* clause. If this matching is not what we want, we can use curly braces to specify the desired *if/else* pairing.

In this code, we have one *if* statement nested within another *if* statement.

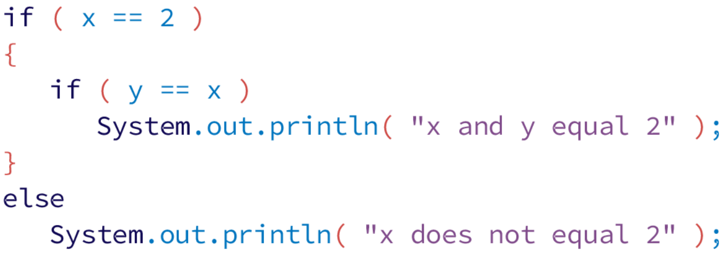


```
if ( x == 2 )
         if ( y == x )
           System.out.println( "x and y equal 2" );
         else
           System.out.println( "x equals 2, but y does not" );
```

Description

Without curly braces, the entire second *if* statement comprises the true block of the first condition `(x == 2)`, and the *else* is paired with the second condition `(y == x)`, because this is the most previous *if* condition that doesn't have an *else*.

However, we can force the *else* clause to be paired with the first condition by using curly braces, as follows:



```
if ( x == 2 )
{
   if ( y == x )
      System.out.println( "x and y equal 2" );
}
else
   System.out.println( "x does not equal 2" );
```

Description

With the curly braces added, the *if* condition `(y == x)`, along with its true block, becomes the complete true block for the condition `(x == 2)`, and the *else* clause now belongs to the first *if* condition `(x == 2)`.

Why can't we just alter the indentation to indicate our meaning? Remember that indentation increases the readability of the code for humans. The compiler ignores indentation and instead follows Java's syntactic rules.

Dangling else

A common error is writing *else* clauses that don't match any *if* conditions. This is called a **dangling else**. For example, the following code, which includes three *else* clauses and only two *if* conditions, will generate this compiler error:

```
'else' without 'if'
```

```
if ( x == 2 )

  if ( y == x )

     System.out.println( "x and y equal 2" );

  else // matches y==x
     System.out.println( "y does not equal 2" );

else // matches x==2
  System.out.println( "x does not equal 2" );

 else // no matching if!
  System.out.println( "x and y are not equal" );
```

Description

COMMON ERROR TRAP

Be sure that all *else* clauses match an *if* condition. Writing *else* clauses that don't match *if* conditions will generate an `'else' without 'if'` compiler error.

For a more complex and real-world example of nested *if* statements, let’s generate a random number between 1 and 10. After we generate the random number, we’ll prompt the user for a guess. First we’ll verify that the guess is between 1 and 10. If it isn’t, we’ll print an error message. Otherwise, we’ll check whether the user has guessed the number. If so, we’ll print a congratulatory message. If the user has not guessed the number, we’ll display the number, then determine whether the guess was close. We’ll define “close” as within three numbers. We’ll print a message informing the user whether the guess was close, then we’ll wish the user better luck next time. The pseudocode for this program looks like this:

```
generate a secret random number between 1 and 10
prompt the user for a guess

if guess is not between 1 and 10
    print message
else
  if guess equals the secret number
    print congratulations
  else
    print the secret number
    if guess is not within 3 numbers
     print "You missed it by a mile!"
    else
     print "You were close."
    print "Better luck next time."
```

Description

This pseudocode uses three *if* statements; the first determines if the guess is within the requested range of numbers. If it isn’t, we print a message. Otherwise, a nested *if* statement tests whether the user has guessed the secret number. If so, we print a congratulatory message. If not, we print the secret number, and our last nested *if* statement determines whether the guess was not within 3 numbers of the secret number. If not, we print “You missed it by a mile!”; otherwise, we print “You were close.” In either case, we print “Better luck next time.”

Example 5.6 is the result of translating this pseudocode into a Java application.

Example 5.6 Nested *if* statements

```
 1 /* Guess a number between 1 and 10
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Random;
 6 import java.util.Scanner;
 7
 8 public class GuessANumber
 9 {
10   public static void main( String [ ] args )
11   {
12     Random random = new Random( );
13     int secretNumber = random.nextInt( 10 ) + 1;
14
15     Scanner scan = new Scanner( System.in );
16
17     System.out.print( "I'm thinking of a number"
18                 + " between 1 and 10. What is your guess? " );
19     int guess = scan.nextInt( );
20
21     if ( guess < 1 || guess > 10 )
22     {
23        System.out.println( "Well, if you're not going to try,"
24                                     + " I'm not playing." );
25     }
26     else
27     {
28       if ( guess == secretNumber )
29            System.out.println( "Hoorah. You win!" );
30       else
31       {
32            System.out.println( "The number was " + secretNumber );
33
34            if ( Math.abs( guess - secretNumber ) > 3 )
35                System.out.println( "You missed it by a mile!" );
36            else
37                System.out.println( "You were close." );
38
39            System.out.println( "Better luck next time." );
40       }
41     }
42   }
43 }
```

Description

On line 34, we used the *abs* method of the *Math* class to determine whether the guess was within three integers of the secret number. By taking the absolute value of the difference between the guess and the secret number, we don’t need to worry about which number is higher than the other; we will always receive a positive difference from the *abs* method.

Figure 5.11 shows the output of a sample run of this program.

Figure 5.11 Output from the *GuessANumber* Program in Example 5.6

Description

5.6 Testing Techniques for *if/else* Statements

When an application uses *if/else* statements, the application's flow of control depends on the user's input or other data values. For one input value, the application may execute the true block, while for another input value, the application may execute the false block. Obviously, running an application only once is no guarantee that the program is correct, because if the true block was executed, then the false block was not executed, and therefore, was not tested. Similarly, if the false block was executed, then the true block was not executed, and therefore was not tested.

To test an application for correctness, we could attempt to test all execution paths. To do this, we devise a **test plan** that includes running the application with different data values designed to execute all the statements in the application.

For example, an application that determines whether an integer is positive or negative might have this code:

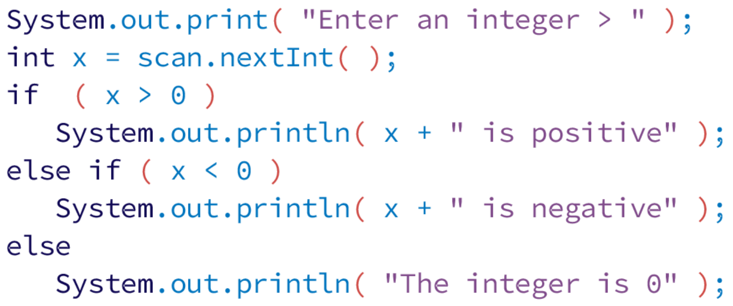
```
System.out.print( "Enter an integer > " );
int x = scan.nextInt( );
if  ( x > 0 )
   System.out.println( x + " is positive" );
else
   System.out.println( x + " is negative" );
```

Description

We could test this code by running the application twice, the first time entering the value 1, and the second time entering the value −1. We see that the results for those two values are correct: 1 is positive and −1 is negative. We have executed all the statements successfully, but can we say for certain that the program is correct? What if we entered the value 0, which is considered neither a positive nor a negative integer? As written, our program determines that 0 is negative, which is incorrect.

We see, then, that testing the true and false blocks is not sufficient; we need to test the condition of the *if/else* statement as well. There are three possibilities: *x* is less than 0, *x* is equal to 0, or *x* is greater than 0. To test the condition, we should run the application with input values that meet these three criteria. So we should run the application one more time with the input value of 0. This will show us that the program is incorrect, because our code identifies 0 as a negative number.

To correct the program, we should add another condition `(x < 0)` so we can separate 0 from the negative numbers. The code would then become:



```
System.out.print( "Enter an integer > " );
int x = scan.nextInt( );
if  ( x > 0 )
   System.out.println( x + " is positive" );
else if ( x < 0 )
   System.out.println( x + " is negative" );
else
   System.out.println( "The integer is 0" );
```

Description

Now if we retest the program with input values –1, 1, and 0, we get correct results for each of these values.

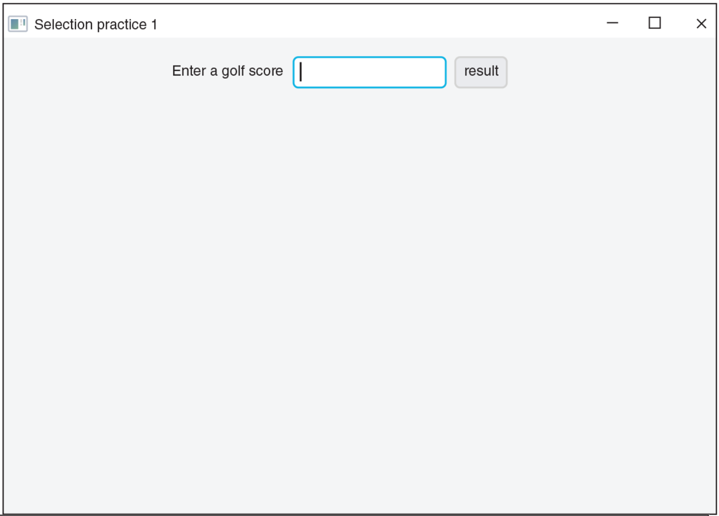
Notice that the test values we chose are –1, 0, and 1. We call these values the *boundary values* because they are the first possible values to cause the conditions (x > 0, and x < 0, and the implicit x == 0 condition) to evaluate to true. If we had incorrectly coded the first condition as x > 1 and we chose, for example, 2, to test our code, our test value of 2 would make the code appear to be correct. However, the code would not work correctly for the value 1. For this reason, we recommend using boundary values to test the execution paths in a program.

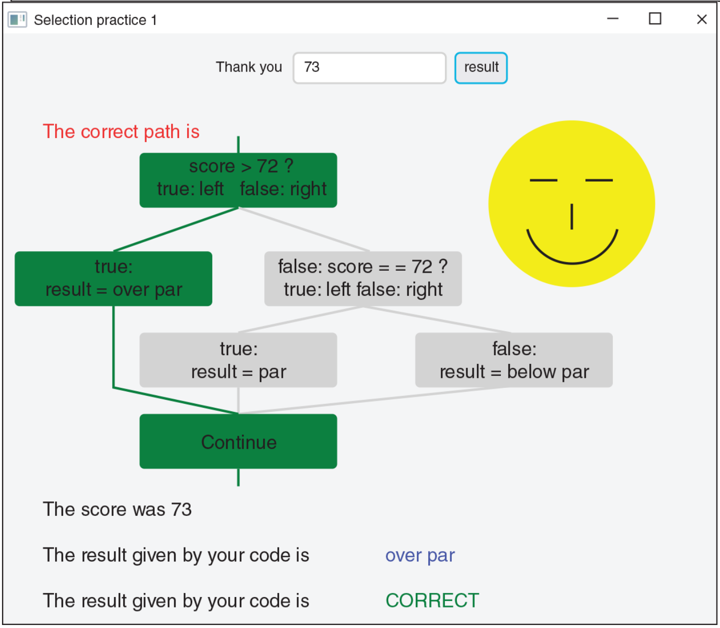
SOFTWARE ENGINEERING TIP

When testing your program, develop input values that test the boundary values for all execution paths and confirm that the logic implements the program specifications.

Another testing method is to treat the program like a black box, that is, as if the program's inner workings are unknown and unknowable to us. We devise our test plan based solely on the specifications of the program and develop input values that test the program logically. Thus, if our specifications are that we should determine whether an integer is positive or negative, we deduce that we should run the program with inputs that are a negative number, a positive number, and the special case, 0.

Both testing methods work together to ensure that a program is correct.


Selection practice 1
Enter a golf score
result


Selection practice 1
result
score > 72 ?
true: left false: right
true:
result = over par
false: score = = 72 ?
true: left false: right
true:
result = par
false:
result = below par
Continue

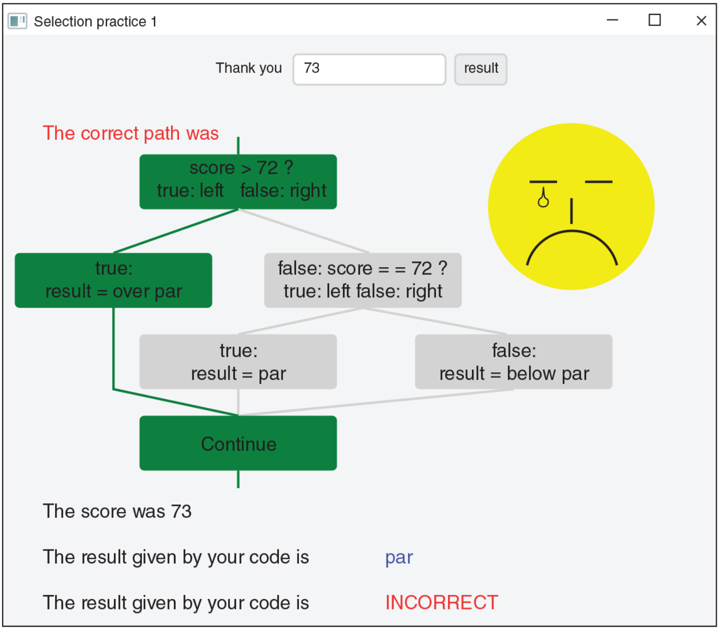
Thank you 73

The correct path is

The score was 73

The result given by your code is over par

The result given by your code is CORRECT


Selection practice 1
result
score > 72 ?
true: left false: right
true:
result = over par
false: score = = 72 ?
true: left false: right
true:
result = par
false:
result = below par
Continue

Thank you 73

The correct path was

The score was 73

The result given by your code is par

The result given by your code is INCORRECT

5.7 Programming Activity 1: Working with *if/else*

In this activity, you will write an *if/else* selection statement to decide how a golfer's score compares to par.

Copy to a folder on your computer all the files from the source code provided with this text for this chapter's Programming Activity 1.

Open the *SelectionPractice1Controller.java* source file. You will add your code to the *workWithIfElse* method. Part of the method has been coded for you. Search for ***** in the source file.

You should be positioned at the code shown in **Example 5.7.**

Example 5.7 The Student Code Portion of Programming Activity 1

```
public void workWithIfElse( int score )
{
        String result = "???";
        // ***** Student code starts here
        // If score is greater than 72, assign "over par" to result
        // If score is equal to 72, assign "par" to result
        // If score is less than 72, assign "below par" to result



        //
        // Student code ends here
        //

        animate( score, result );
}
```

Description

Where indicated in the code, you should write an *if/else* statement to perform the following function:

- In the method header of the method *workWithIfElse*, you see `( int score )`.
- The *int* variable *score* represents a golf score. This variable will be an input from the user; the text box that allows the user to enter the score has already been coded for you and stores the user's input in the variable *score*, which is available to your code as a parameter of the *workWithIfElse* method. Do not declare the variable *score* inside the method; just use it.
- We want to know if the golf score is "over par," "par," or "below par." Par is 72.
- Inside the *if/else* statement, you need to assign a value to the *String* variable named *result*, as follows:

  > If *score* is higher than 72, then assign "over par" to *result*; if score is exactly 72, assign "par" to *result*; and if score is lower than 72, assign "below par" to *result*.

- You do not need to write the code to call the method *animate*; that part of the code has already been written for you.

Animation: The application window will display the correct path of the *if/else* statement (in green), which may or may not be the same as your path, depending on how you coded the *if/else* statement. The animation will also assess your result, that is, the value of the variable *result*, and give you feedback on the correctness of your result.

To test your code, compile *SelectionPractice1Controller.java* and run *SelectionPractice1Application* and enter an integer in the text box. Try the following input values for *score*: 71, 72, and 73. Be sure your code produces the correct result for all input values.

When the program begins, you will see a graphics window with the text box of **Figure 5.12**, prompting you for an integer value.

Figure 5.12 The Beginning of the Application

Description

Figure 5.13 demonstrates the correct code path when the input value is 73 and assesses that the student’s code is correct.

Figure 5.13 A Correct *if/else* Statement

Description

Figure 5.14 again demonstrates the correct code path when the input value is 73, but in this case, the student’s code is incorrect.

Figure 5.14 An Incorrect *if/else* Statement

Description

DISCUSSION QUESTIONS

1. How many conditions did you use in the complete *if/else* statement?

2. Your code should be correct if the application gets correct results for the input values 71, 72, and 73. Explain why.

```
d1 = 1.0999999999999999
d2 = 1.1
d1 and d2 are not equal

piF = 3.1415927
pid = 3.141592653589793
piF and piD are not equal
```

```
d1 = 1.0999999999999999
d2 = 1.1
d1 and d2 are considered equal

piF = 3.1415927
pid = 3.141592653589793
piF and piD are considered equal
```

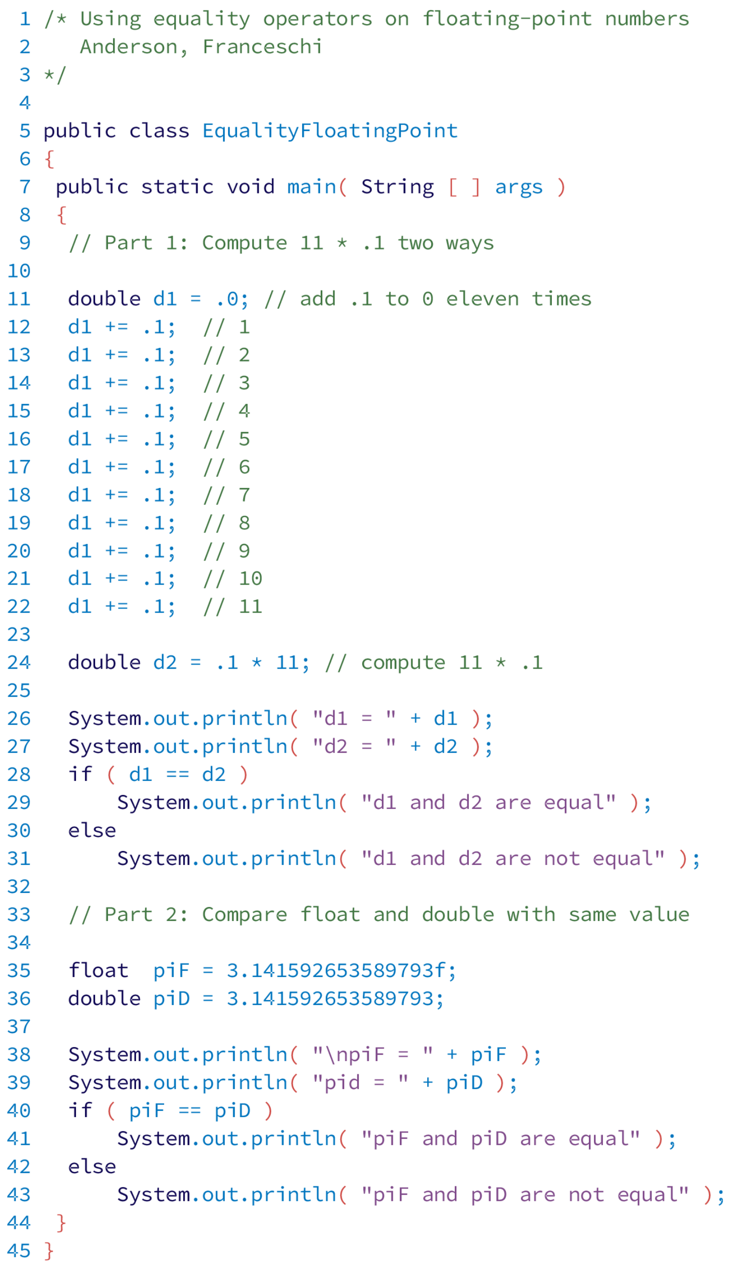
```
d1 = 1.1
d2 = 1.1
d1 and d2 are equal
```

5.8 Comparing Floating-Point Numbers

Java's *floats* and *doubles* are stored using IEEE 754 standard format. Because a finite number of bits (32 or 64) is used to store floating-point numbers, not all real-world values can be represented. As a result, minor rounding errors can be introduced when arithmetic is performed. That said, it is not advisable to simply rely on the equality operators to compare floating-point numbers.

Let's take a look at **Example 5.8**, which computes 11 * .1 two ways. First, at line 11, we assign .0 to a *double* variable, *d1*, and at lines 12–22 we add .1 to *d1* eleven times. Then, at line 24, we declare a second *double* variable, *d2*, and assign it the result of multiplying .1 times 11. We would expect, then, that *d1* and *d2* would have the same value. Not so, as the output of the program shows in **Figure 5.15**.

x**Example 5.8** Using the Equality Operator to Compare Floating-Point Numbers



```
/* Using equality operators on floating-point numbers
   Anderson, Franceschi
*/

public class EqualityFloatingPoint
{
 public static void main( String [ ] args )
 {
  // Part 1: Compute 11 * .1 two ways

  double d1 = .0; // add .1 to 0 eleven times
  d1 += .1;  // 1
  d1 += .1;  // 2
  d1 += .1;  // 3
  d1 += .1;  // 4
  d1 += .1;  // 5
  d1 += .1;  // 6
  d1 += .1;  // 7
  d1 += .1;  // 8
  d1 += .1;  // 9
  d1 += .1;  // 10
  d1 += .1;  // 11

  double d2 = .1 * 11; // compute 11 * .1

  System.out.println( "d1 = " + d1 );
  System.out.println( "d2 = " + d2 );
  if ( d1 == d2 )
      System.out.println( "d1 and d2 are equal" );
  else
      System.out.println( "d1 and d2 are not equal" );

  // Part 2: Compare float and double with same value

  float  piF = 3.141592653589793f;
  double piD = 3.141592653589793;

  System.out.println( "\npiF = " + piF );
  System.out.println( "pid = " + piD );
  if ( piF == piD )
      System.out.println( "piF and piD are equal" );
  else
      System.out.println( "piF and piD are not equal" );
 }
}
```

Description

Figure 5.15 Output from Example 5.8

Description

REFERENCE POINT

Binary representation of floating-point numbers is discussed in Appendix E.

We can also see the effects of rounding when comparing a *float* to a *double*. For example, at lines 35 and 36 of Example 5.8, we assign the same floating-point number (PI) to a double variable *piD* and to a float variable *piF*, then compare the two values at line 40. As we can see from the output in Figure 5.15, they do not compare as equal. The reason is that double-precision floating-point numbers are able to store a larger number of significant digits than single-precision floating-point numbers.

Instead of using the equality operator to compare floating-point numbers, it's better to compare the absolute value of the difference to a small value, called a *threshold*. The value of the threshold should be the difference we can tolerate and still consider the numbers equal. Let's redo Example 5.8. Instead of using the equality operator, we'll use the *Math.abs* method to compute a difference between the two numbers and compare the difference to a threshold value. We'll set the threshold at .0001, meaning that if the numbers differ by less than .0001, we'll consider them equal. The results of this approach are shown in **Example 5.9** and the output is given in **Figure 5.16**.

Example 5.9 Comparing Floating-Point Numbers Using a Threshold

```
/* Using a threshold to compare floating-point numbers
   Anderson, Franceschi
*/

public class ComparingFloatingPoint
{
 public static void main( String [ ] args )
 {
   final double THRESHOLD = .0001;

   // Part 1: Compute 11 * .1 two ways
   double d1 = .0; // add .1 to 0 eleven times
   d1 += .1;  // 1
   d1 += .1;  // 2
   d1 += .1;  // 3
   d1 += .1;  // 4
   d1 += .1;  // 5
   d1 += .1;  // 6
   d1 += .1;  // 7
   d1 += .1;  // 8
   d1 += .1;  // 9
   d1 += .1;  // 10
   d1 += .1;  // 11

   double d2 = .1 * 11; // compute 11 * .1

   System.out.println( "d1 = " + d1 );
   System.out.println( "d2 = " + d2 );
   if ( Math.abs( d1 - d2 ) < THRESHOLD )
        System.out.println( "d1 and d2 are considered equal" );
   else
        System.out.println( "d1 and d2 are not equal" );

   // Part 2: Compare float and double with same value
   float  piF = 3.141592653589793f;
   double piD = 3.141592653589793;

   System.out.println( "\npiF = " + piF );
   System.out.println( "piD = " + piD );
   if ( Math.abs( piF - piD ) < THRESHOLD )
        System.out.println( "piF and piD are considered equal" );
   else
        System.out.println( "piF and piD are not equal" );
 }
}
```

Description

Figure 5.16 Output of Example 5.9

Description

When we need exact precision in calculations with decimal numbers, we can use the *BigDecimal* class in the Java Class Library. The *BigDecimal* class, which is in the *java.math* package, provides methods that perform addition, subtraction, multiplication, and division of *BigDecimal* objects so that the results are exact, without the rounding errors caused by floating-point operations. **Table 5.10** shows a constructor of the *BigDecimal* class and several useful methods for performing calculations and comparing *BigDecimal* objects.

The *BigDecimal* Class API

BigDecimal Class Constructor Summary

`BigDecimal( String ddd )`

creates a *BigDecimal* object equivalent to the decimal number expressed as a *String*

***BigDecimal* Class Method Summary**

Return value	**Method name and argument list**
`BigDecimal`	`add( BigDecimal num )` returns a *BigDecimal* object equal to the current *BigDecimal* object plus *num*
`BigDecimal`	`subtract( BigDecimal num )` returns a *BigDecimal* object equal to the current *BigDecimal* object minus *num*
`BigDecimal`	`multiply( BigDecimal num )` returns a *BigDecimal* object equal to the current *BigDecimal* object times *num*
`BigDecimal`	`divide( BigDecimal num )` returns a *BigDecimal* object equal to the current *BigDecimal* object divided by *num*
`int`	`compareTo( BigDecimal`

	`num )` returns 0 if the current *BigDecimal* object is equal to *num*; –1 if the current *BigDecimal* object is less than *num*; and 1 if the current *BigDecimal* object is greater than *num*

Description

REFERENCE POINT

You can read more about the *BigDecimal* class on Oracle's Java website *www.oracle.com/technetwork/java*.

In **Example 5.10**, we perform the same calculations as in Example 5.9, but we use *BigDecimal* objects instead of *doubles*. On lines 11 and 12, we instantiate two *BigDecimal* objects: *d1* will hold the sum and is initialized to 0.0; *pointOne* is assigned the value 0.1 and will be repeatedly added to *d1*. Then on lines 16–26, we call the *add* method to add 0.1 to *d1* 11 times. We instantiate two more *BigDecimal* objects on lines 29 and 30, then call the *multiply* method to multiply 0.1 * 11. On line 35, we compare the resulting *BigDecimal* objects by calling the *compareTo* method, and find that the two results are in fact equal. The output of Example 5.10 is shown in **Figure 5.17**.

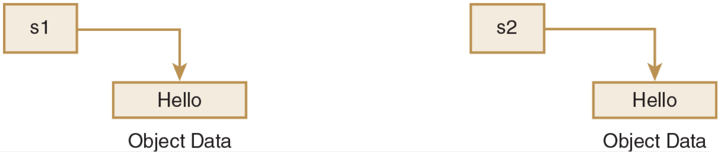
Example 5.10 Comparing Floating-Point Numbers Using *BigDecimal*

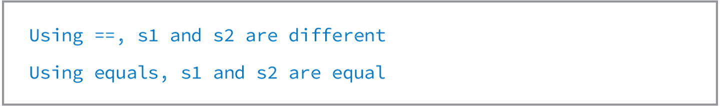
```
/* Using BigDecimal to compute precise decimal numbers
   Anderson, Franceschi
*/

import java.math.BigDecimal;

public class UsingBigDecimal
{
 public static void main( String [ ] args )
 {
   BigDecimal d1 = new BigDecimal( "0.0" );
   BigDecimal pointOne = new BigDecimal( "0.1" );

   // Compute 11 * .1 two ways
   // add .1 to d1 eleven times
   d1 = d1.add( pointOne ); // 1
   d1 = d1.add( pointOne ); // 2
   d1 = d1.add( pointOne ); // 3
   d1 = d1.add( pointOne ); // 4
   d1 = d1.add( pointOne ); // 5
   d1 = d1.add( pointOne ); // 6
   d1 = d1.add( pointOne ); // 7
   d1 = d1.add( pointOne ); // 8
   d1 = d1.add( pointOne ); // 9
   d1 = d1.add( pointOne ); // 10
   d1 = d1.add( pointOne ); // 11

   // multiply .1 * 11
   BigDecimal d2 = new BigDecimal( "0.1" );
   BigDecimal eleven = new BigDecimal( "11" );
   d2 = d2.multiply( eleven );

   System.out.println( "d1 = " + d1 );
   System.out.println( "d2 = " + d2 );
   if ( d1.compareTo( d2 ) == 0 )
        System.out.println( "d1 and d2 are equal" );
   else
        System.out.println( "d1 and d2 are not equal" );
 }
}
```

Description

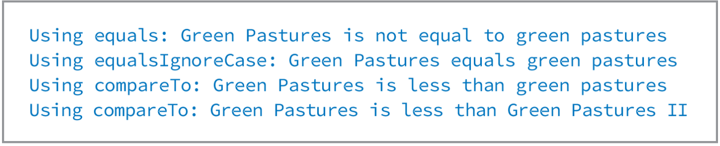
Figure 5.17 Output of Example 5.10

Description





```
Using ==, s1 and s2 are different

Using equals, s1 and s2 are equal
```



```
Using equals: Green Pastures is not equal to green pastures
Using equalsIgnoreCase: Green Pastures equals green pastures
Using compareTo: Green Pastures is less than green pastures
Using compareTo: Green Pastures is less than Green Pastures II
```

5.9 Comparing Objects

5.9.1 The *equals* Method

Often, we want to compare whether two objects are equal; typically, we will say that two objects are equal if they have the same data. If we use the equality operator (==) to compare object references, however, we are comparing the value of the object references. In other words, we are comparing whether the object references point to the same object, that is, the same memory location. To compare object data, we need to use the *equals* method, which all classes inherit from the *Object* class. Many classes provide a custom version of the *equals* method. The API of the *equals* method is the following:

```
public boolean equals( Object ob )
```

Description

Typically, the *equals* method returns *true* if the data in the parameter object matches the data in the object for which the method was called.

The program in **Example 5.11** creates the two *String* object references and objects shown in **Figure 5.18**. The program compares the object references using the equality operator and then compares the object data using the *equals* method. The output from this program is shown in **Figure 5.19**.

Example 5.11 Comparing Object Data

```
 1 /* Comparing Object References and Data
 2    Anderson, Franceschi
 3 */
 4 public class ComparingObjects
 5 {
 6   public static void main( String [] args )
 7   {
 8     String s1 = new String( "Hello" );
 9     String s2 = new String( "Hello" );
10
11     if( s1 == s2 )
12       System.out.println( "Using ==, s1 and s2 are equal" );
13     else
14       System.out.println( "Using ==, s1 and s2 are different" );
15
16     if( s1.equals( s2 ) )
17       System.out.println( "Using equals, s1 and s2 are equal" );
18     else
19       System.out.println( "Using equals, s1 and s2 are different" );
20   }
21 }
```

Description

Figure 5.18 *String* Objects and References

Description

Figure 5.19 Output from Example 5.11

Description

Lines 8 and 9 instantiate two *String* objects with the same data.

COMMON ERROR TRAP

Do not use the equality operators to compare object data; instead, use the *equals* method.

At line 11, when we compare *s1* and *s2* using the equality operator, the result is *false*, because the object references *s1* and *s2* point to two different objects.

We get different results using the *equals* method. When line 16 compares *s1* and *s2* using the *equals* method, the result is *true*, because *s1* and *s2* have identical data.

5.9.2 *String* Comparison Methods

In addition to the *equals* method, the *String* class provides two other methods, *equalsIgnoreCase* and *compareTo*, for comparing the values of *Strings*. These methods, along with the *equals* method are summarized in **Table 5.11**.

Comparison Methods of the *String* Class

String Methods for Comparing *String* Values

Return value	**Method name and argument list**
`boolean`	`equals ( String str )` compares the value of two *Strings*. Returns *true* if the *Strings* are equal; *false* otherwise.
`boolean`	`equalsIgnoreCase ( String str )` compares the value of two *Strings*, treating upper and lowercase characters as equal. Returns *true* if the *Strings* are equal; *false* otherwise.
`int`	`compareTo ( String str )` compares the value of the two *Strings* in lexicographic order. If the *String* object is less than the *String* argument, *str*, a negative integer is returned. If the *String* object is greater than the *String* argument, a positive number is returned; if the two *Strings* are equal, 0 is returned.

Description

The *equalsIgnoreCase* method is similar to the *equals* method, except that it is insensitive to case. Thus, the *equalsIgnoreCase* method returns *true* if the two *String* objects have the same sequence of characters, regardless of capitalization. For example, the *equalsIgnoreCase* method considers *ABC, AbC*, and *abc* to be equal.

REFERENCE POINT

The first 128 Unicode values are given in Appendix C.

The *compareTo* method returns an integer value, rather than a *boolean* value. The *compareTo* method's return value represents whether the *String* object is less than, equal to, or greater than the *String* argument passed to the *compareTo* method. The *compareTo* method uses lexicographic order—the Unicode collating sequence—to compare the *Strings*. Using the Unicode collating sequence means that a character with a lower Unicode numeric value is considered less than a character with a higher Unicode numeric value. Thus, an *a* is lower than a *b*; an *A* is lower than a *B*; and *0* is lower than *1*.

The *compareTo* method scans the two *Strings* from left to right. If it finds different characters in the same position in the two *Strings*, it immediately returns an integer value representing the difference between the Unicode values of those characters. For example, the distance between *a* and *c* is −2; the distance between *K* and *F* is 5.

If the *Strings* differ in length, but the characters they have in common are identical, then the *compareTo* method returns the difference in the length of the *Strings*.

In most cases, however, the exact return value is not important; it is sufficient to know whether the *String* object is less than, greater than, or equal to the *String* argument. In other words, all that we usually need to know is whether the return value is positive, negative, or 0.

Example 5.12 demonstrates how these methods can be used in a Java application to compare *Strings*. The output of the program is shown in **Figure 5.20**.

Example 5.12 Comparing Strings

```
 1 /* Demonstration of the String comparison methods
 2    Anderson, Franceschi
 3 */
 4
 5 public class ComparingStrings
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     String title1 = "Green Pastures";
10     String title2 = "Green Pastures II";
11     String title3 = "green pastures";
12
13     System.out.print( "Using equals: " );
14     if ( title1.equals( title3 ) )
15       System.out.println( title1 + " equals " + title3 );
16     else
17       System.out.println( title1 + " is not equal to " + title3 );
18
19     System.out.print( "Using equalsIgnoreCase: " );
20     if ( title1.equalsIgnoreCase( title3 ) )
21       System.out.println( title1 + " equals " + title3 );
22     else
23       System.out.println( title1 + " is not equal to " + title3 );
24
25     System.out.print( "Using compareTo: " );
26     if ( title1.compareTo( title3 ) > 0 )
27       System.out.println( title1 + " is greater than " + title3 );
28     else if ( title1.compareTo ( title3 ) < 0 )
29       System.out.println( title1 + " is less than " + title3 );
30     else
31       System.out.println( title1 + " is equal to " + title3 );
32
33     System.out.print( "Using compareTo: " );
34     if ( title1.compareTo( title2 ) > 0 )
35       System.out.println( title1 + " is greater than " + title2 );
36     else if ( title1.compareTo( title2 ) < 0 )
37       System.out.println( title1 + " is less than " + title2 );
38     else
39       System.out.println( title1 + " is equal to " + title2 );
40   }
41 }
```

Description

Figure 5.20 Output from Example 5.12

Description

In Example 5.12, we define three similar *Strings*: *title1 (Green Pastures), title2 (Green Pastures II)*, and *title3 (green pastures)*. When we compare *title1, Green Pastures, to title3, green pastures*, using the *equals* method (line 14), the result is *false*, because the *Strings* do not match in case. When we perform the same comparison using the *equalsIgnoreCase* method (line

20), however, the result is *true*, because except for capitalization, these two *Strings* are identical in character sequence and length.

Using the *compareTo* method (line 34), *Green Pastures* evaluates to less than *Green Pastures II*. Although all the characters of the first *String* are found in the second *String* in the same order, the first *String* has fewer characters than the second *String*. The reason that *Green Pastures* evaluates to less than *green pastures* (line 26) is not so obvious—until we look at the Unicode character chart. The capital letters have lower numeric values than the lowercase letters, so a capital *G* is less than a lowercase *g*.

```
Enter an integer > 12
12 is even.
```

```
Enter an integer > 13
13 is odd.
```

5.10 The Conditional Operator (?:)

The conditional operator (?:), while not a statement in itself, can be used in expressions. It evaluates a condition and contributes one of two values to the expression based on the value of the condition. The conditional operator is especially useful for handling invalid input and for outputting similar messages. The syntax of the conditional operator is shown here:

```
( condition ? expression1 : expression2 )
```

Description

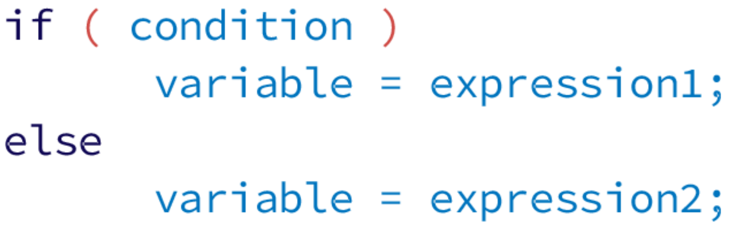
The value of an expression containing a conditional operator is determined by evaluating the condition, which is any expression that evaluates to *true* or *false*. If the condition evaluates to *true, expression1* becomes the value of the expression; if the condition evaluates to *false, expression2* becomes the value of the expression.

When assigning the result of that expression to a variable, the statement:

```
variable = ( condition ? expression1 : expression2 );
```

Description

is equivalent to

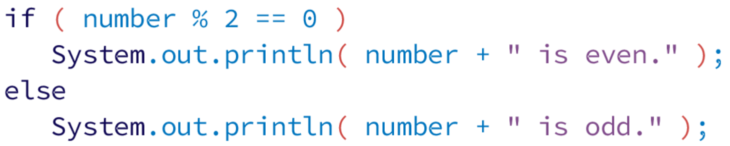


```
if ( condition )
       variable = expression1;
else
       variable = expression2;
```

Description

Some programmers like to use the conditional operator because it enables them to write compact code; other programmers feel that an *if/else* sequence is more readable.

Suppose we want to determine whether a number is even or odd. We know that if we divide a number by 2 and the remainder is 0, the number is even. If the remainder is 1, the number is odd. We could do this using the following *if/else* statement:



```
if ( number % 2 == 0 )
   System.out.println( number + " is even." );
else
   System.out.println( number + " is odd." );
```

Description

Notice that the only difference in the output is the last word.

Example 5.13 shows how we can perform the same processing using the conditional operator.

Example 5.13 Using the Conditional Operator

```
 1 /* Using the conditional operator
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6
 7 public class OddOrEven
 8 {
 9  public static void main( String [ ] args )
10  {
11   Scanner scan = new Scanner( System.in );
12
13   System.out.print( "Enter an integer > " );
14   int number = scan.nextInt( );
15
16   String outcome = ( number % 2 == 0 ? "even." : "odd." );
17
18   System.out.println( number + " is " + outcome );
19
20  }
21 }
```

Description

In lines 13 and 14, we ask the user to input an integer. In line 16, we use the conditional operator to determine whether the integer is even or odd, assigning the result to the *String outcome*. Then in line 18, we print the number, the word “ is ” and then the value of *outcome*. **Figure 5.21** shows the output from **Example 5.13** when the user enters an even number, and Figure 5.22 shows the output when the user enters an odd number.

Figure 5.21 Output of Example 5.13 When the Input Is Even

Description

Figure 5.22 Output of Example 5.13 When the Input Is Odd

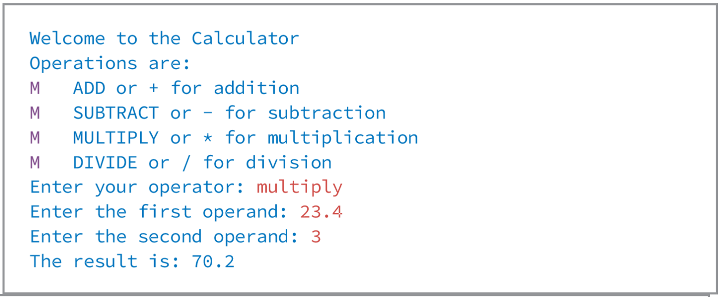
Description

Table 5.12, Operator Precedence, shows that the conditional operator is low in precedence, being just above the assignment operators.

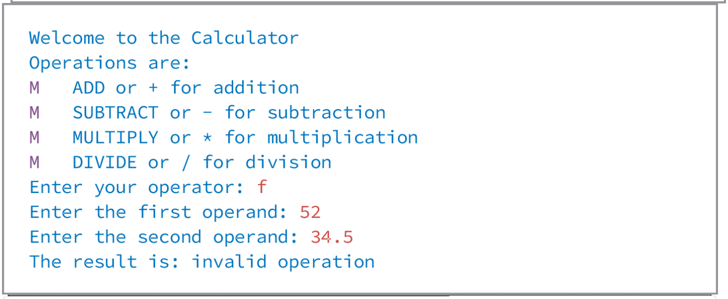
Operator Precedence

Operation Hierarchy	Order of Same-Statement Evaluation	Operation
( )	left to right	parentheses for explicit grouping
++, --	right to left	shortcut postincrement
++, --, !	right to left	shortcut preincrement, logical unary NOT
*, /, %	left to right	multiplication, division, modulus
+, -	left to right	addition or *String* concatenation, subtraction
<, <=, >, >=	left to right	relational operators: less than, less than or equal to, greater than, greater than or equal to
==, !=	left to right	equality operators: equal to and not equal to
&&	left to right	logical AND
||	left to right	logical OR
?:	left to right	conditional operator
=, +=, -=, *=, /=, %=	right to left	assignment operator and shortcut assignment operators

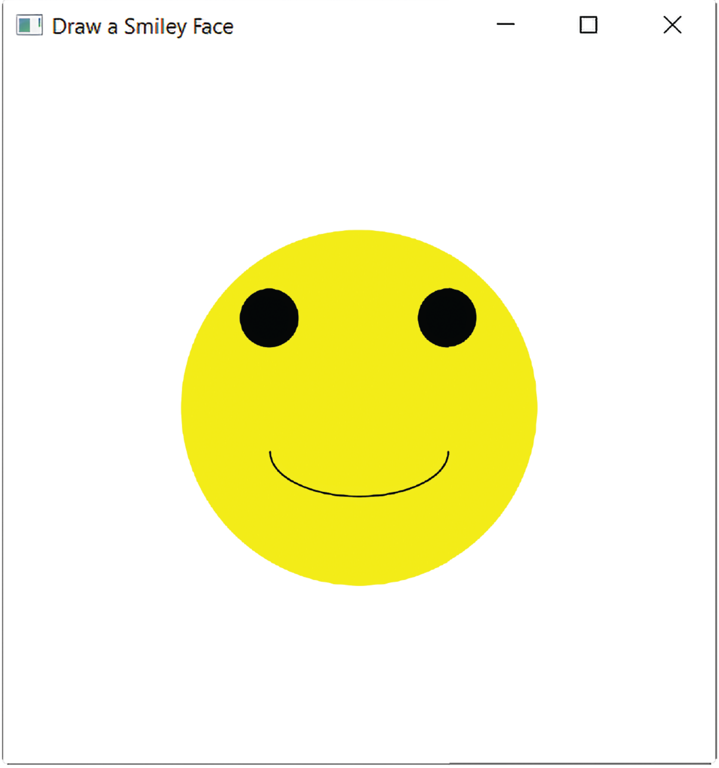
Description



```
Welcome to the Calculator
Operations are:
M   ADD or + for addition
M   SUBTRACT or - for subtraction
M   MULTIPLY or * for multiplication
M   DIVIDE or / for division
Enter your operator: multiply
Enter the first operand: 23.4
Enter the second operand: 3
The result is: 70.2
```



```
Welcome to the Calculator
Operations are:
M   ADD or + for addition
M   SUBTRACT or - for subtraction
M   MULTIPLY or * for multiplication
M   DIVIDE or / for division
Enter your operator: f
Enter the first operand: 52
Enter the second operand: 34.5
The result is: invalid operation
```



5.11 The *switch* Statement

The *switch* statement can be used instead of an *if/else if* statement for selection when the condition consists of comparing the value of an expression to constant integers (*byte, short*, or *int*), characters (*char*), or *Strings*. The latest syntax of the *switch* statement, introduced with Java 14, is as follows:

```
switch( expression )
{
   case constant1A, constant1B, … -> statement or block;
   case constant2A, constant2B, … -> statement or block;
   …
   default -> statement or block; // optional (but often needed)
}
```

Description

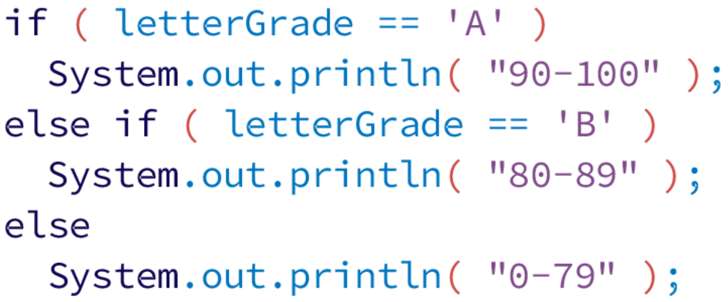
The expression is first evaluated; then its value is compared to the *case* constants in order. When a match is found, the statement or block associated with that *case* constant is executed. Then, execution jumps to the end of the *switch* statement. Cases must be exhaustive. Thus, unless the expression can match a finite list of values, a *default* label is required.

For example, the following snippet:

```
switch ( letterGrade )
{
   case 'A' -> System.out.println( "90-100" );
   case 'B' -> System.out.println( "80-89" );
   default -> System.out.println( "0-79" );
}
```

Description

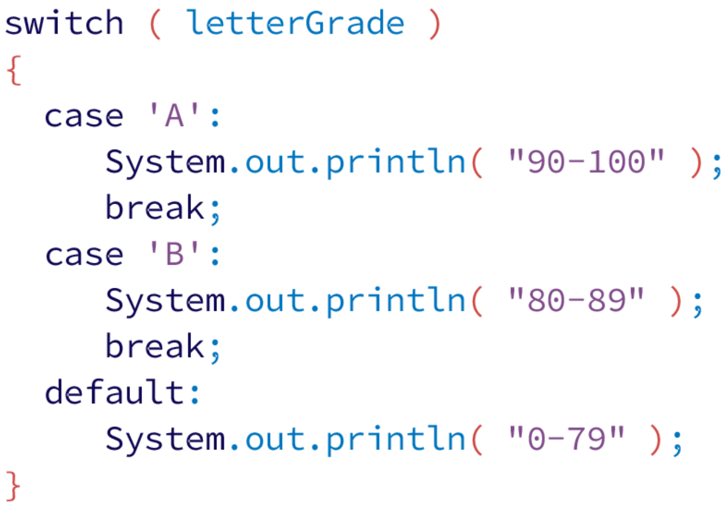
is equivalent to this snippet, using *if/else if/else*:



```
if ( letterGrade == 'A' )
  System.out.println( "90-100" );
else if ( letterGrade == 'B' )
  System.out.println( "80-89" );
else
  System.out.println( "0-79" );
```

Description

Using the old *switch* syntax (Java 13 and before, but still supported at the time of publication of this book), the equivalent of the above snippet is:



```
switch ( letterGrade )
{
  case 'A':
    System.out.println( "90-100" );
    break;
  case 'B':
    System.out.println( "80-89" );
    break;
  default:
    System.out.println( "0-79" );
}
```

Description

Furthermore, we can use a **switch expression** to assign a value to a variable depending on various scenarios. A *switch* expression uses a *switch* statement. The statement that follows a *case* label is, in this case, a value that is assigned to the variable. If the value that we want to assign to a variable requires calculations that involve several statements, we can use a block. Inside that block, the value that follows the *yield* keyword is the value that is assigned to the variable.

```
datatype identifier = switch( expression )
{
  case constant1A, constant1B, … -> expression or block with yield;
  case constant2A, constant2B, … -> expression or block with yield;
  …
  // optional default (but often needed)
  default -> expression or block with yield;
}
```

Description

Let’s look at how a *switch* expression can be used to implement a simple calculator. We first prompt the user for two numbers on which they want to perform a calculation, and then the operation they want to perform. We let them enter either the words ADD, SUBTRACT, MULTIPLY, or DIVIDE, or the symbol for the operation (1, 2, *, or /). We can use a *switch* expression to determine the outcome of the selected operation. **Example 5.14** shows the code for our simple calculator.

Example 5.14 A Simple Calculator

```java
/* A simple calculator
   Anderson, Franceschi
*/

import java.text.DecimalFormat;
import java.util.Scanner;

public class Calculator
{
  public static void main( String [ ] args )
  {
    Scanner scan = new Scanner( System.in );

    // set up the output format of the result
    DecimalFormat twoDecimals = new DecimalFormat( "#,###.##" );

    // print a welcome message
    System.out.println( "Welcome to the Calculator" );

    //  define and print a menu, then prompt for the operation
    String menu = """
      Operations are:
      \tADD or + for addition
      \tSUBTRACT or - for subtraction
      \tMULTIPLY or * for multiplication
      \tDIVIDE or / for division""";
    System.out.println( menu );
    System.out.print( "Enter your operator: " );
    String operation = scan.next( );
    operation = operation.toUpperCase( );

    // read the two operands
    System.out.print( "Enter the first operand: " );
    double fp1 = scan.nextDouble( );
    System.out.print( "Enter the second operand: " );
    double fp2 = scan.nextDouble( );

    // perform the operation
    String result = switch ( operation )
    {
      case "ADD", "+" -> twoDecimals.format( fp1 + fp2 );
      case "SUBTRACT", "-" -> twoDecimals.format( fp1 - fp2 );
      case "MULTIPLY", "*" -> twoDecimals.format( fp1 * fp2 );
      case "DIVIDE", "/" ->
      {
        if ( fp2 == 0.0 )
          yield "Dividing by 0 is not allowed";
        else
          yield twoDecimals.format( fp1 / fp2 );
      }
      default -> "invalid operation";
    };

    // output the result of the operation
    System.out.println( "The result is: " + result );
  }
}
```

Description

At lines 20–30, we define a menu using a text block, output it, ask the user to choose an operation, and convert it to upper case. At lines 32–36, we ask the user to input the two operands for the operation.

We are now ready to process the user's inputs, by using a *switch* statement with the user's selected operation as the *switch expression*. Note that because we switch on a *String*, the *default* case is required since cases must be exhaustive.

We determine which operation the user has selected by providing a *case* statement for each possible operation. We can use uppercase words as the *case* constants, because we have converted the input to uppercase. We handle the situation where the user has entered a mathematical symbol instead of a word by adding each symbol to the list of *case* constants for each operation. For example, if the user enters ADD (in any combination of uppercase and lowercase letters) or a plus sign (1), the input will match one of our *case* constants at line 41. We assign the result to the variable *result* at line 39. If we are performing a division (line 44), we need to check whether the divisor is 0 before performing division (line 46). Because we use an *if/else* statement for this with two possible outcomes, we use a block. Thus, we need to use the *yield* keyword to assign the resulting value to *result* at line 47. Although we discussed earlier in the chapter that we should compare floating-point numbers by comparing the difference between the two numbers with a threshold value, in this case we care only if the second operand is exactly 0, so we can safely compare its value to 0.0. If the second operand is not 0.0 (line 48), we perform the division at line 49 and format the result. Note that we could have used the ?: operator instead of the *if/else* statement and avoided using a block. This is left as an exercise. What if the user doesn't enter any of the valid words or mathematical symbols? This is handled by the *default* case, allowing us to assign an error message to *result* (line 51).

Figure 5.23 shows the output from Example 5.14 when the user selects multiplication, and **Figure 5.24** shows the output when the user enters an unsupported operation.

Figure 5.23 The Calculator Performing Multiplication

Description

Figure 5.24 The Calculator with an Invalid Entry for the Operation

Description

We'll create a graphical application that draws a smiley face depending on a value for a rating. In **Example 5.15**, we ask the user for a rating between 1 and 5 (lines 24–27). We define constant values that we will use to draw the smiley face at line 22. At lines 29–36, we draw the face and the eyes. Then, we use a *switch* statement (lines 38–50) to draw the mouth depending on the rating.

Example 5.15 Drawing a Smiley Face

```java
/* Drawing a smiley face
   Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.scene.shape.ArcType;
import javafx.stage.Stage;

import java.util.Scanner;

public class SmileyFaceDraw extends Application
{
  @Override
  public void start( Stage stage )
  {
    // set up window title and size
    GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
                  stage, "Draw a Smiley Face", 400, 400 );

    final int X = 100, Y = 100, RADIUS = 100;

    // ask the user for a rating
    Scanner scan = new Scanner( System.in );
    System.out.print( "Enter a rating (1 to 5) > " );
    int rating = scan.nextInt( );

    // Draw the face and the eyes
    gc.setFill( Color.YELLOW );
    gc.fillOval( X, Y, 2 * RADIUS, 2 * RADIUS );
    gc.setFill( Color.BLACK );
    gc.fillOval( X + RADIUS / 3, Y + RADIUS / 3,
                 RADIUS / 3, RADIUS / 3 );
    gc.fillOval( X + 4 * RADIUS / 3, Y + RADIUS / 3,
                 RADIUS / 3, RADIUS / 3 );

    // Draw the smile
    switch ( rating )
    {
      case 1, 2 ->
        gc.strokeArc( X + RADIUS / 2, Y + RADIUS,
                      RADIUS, RADIUS / 2, 0, 180, ArcType.OPEN );
      case 4, 5 ->
        gc.strokeArc( X + RADIUS / 2, Y + RADIUS,
                      RADIUS, RADIUS / 2, 0, -180, ArcType.OPEN );
      default ->
        gc.strokeLine( X + RADIUS / 2, Y + 4 * RADIUS / 3,
                       X + 3 * RADIUS / 2, Y + 4 * RADIUS / 3 );
    } // end switch
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description

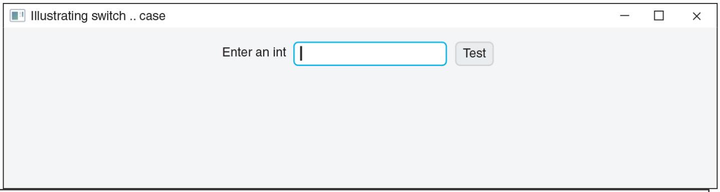
We draw a frown at lines 42–43 if the rating is 1 or 2 (line 41). We draw a smile at lines 45–46 if the rating is 4 or 5 (line 44), and a straight line at lines 48–49 otherwise (rating of 3). Note that because we *switch* on an *int*, the *default* case is mandatory.

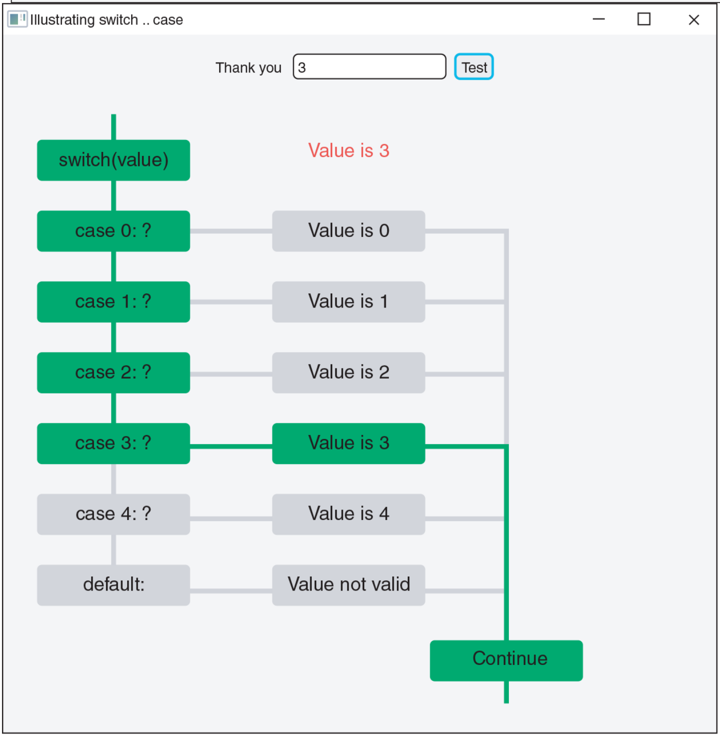
The output when the user input is *2* is shown in **Figure 5.25**.

Figure 5.25 The Smiley Face with User Input of 2

with these end-of-chapter questions

5.14.1	Multiple Choice Exercises Question 9
5.14.2	Reading and Understanding Code Questions 14, 16, 17, 18, 19
5.14.5	Debugging Area Questions 39, 41
5.14.6	Write a Short Program Questions 44, 45, 47
5.14.8	Technical Writing Question 53


Illustrating switch .. case
Enter an int
Test


Illustrating switch .. case
Thank you
3
Test
Value is 3
switch(value)
case 0: ?
Value is 0
case 1: ?
Value is 1
case 2: ?
Value is 2
case 3: ?
Value is 3
case 4: ?
Value is 4
default:
Value not valid
Continue

5.12 Programming Activity 2: Using the *switch* Statement

In this activity, you will write a *switch* statement that selects a path depending on an input value. The framework will animate your code so that you can watch the path that the code takes in the *switch* block.

Copy to a folder on your computer all the files in this chapter's Programming Activity 2 folder in the supplied code accompanying this book.

Search for five stars (*****) in the

SelectionPractice2Controller.java source code to find where to add your code. The five stars are above the method *workWithSwitch* (the method header has already been coded for you).

You should be positioned at the code shown in **Example 5.16**.

Example 5.16 The Student Code Portion of Programming Activity 2

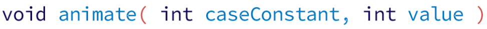
```
// ***** 1 student writes this method
public void workWithSwitch( int value )
{
  //
  // Student code starts here
  //



  //
  // Student code ends here
  //



}
// end of workWithSwitch
```

Description

Where indicated in the code, write a *switch* statement, as follows:

- In the method header of the method *workWithSwitch*, you see `( int value )`. The *int* variable *value* represents the input from the user; the text box that allows the user to input the score has already been coded for you. This variable, *value*, is the value entered by the user and should be used as the condition for the *switch* statement; it is available to your code as a parameter of the *workWithSwitch* method. Do not declare the variable *value* inside the method; just use it.

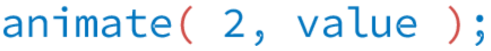
- Write *case* statements for the following integer constants: *0, 1, 2, 3, 4*, as well as a *default* statement.
- Within each *case* statement, you should do two things:
    - Print a message to the screen indicating which value was input. The message for the *default* case should indicate that the input value is not one of the valid values.
    - Call the *animate* method. The API for the *animate* method is



```
void animate( int caseConstant, int value )
```

Description

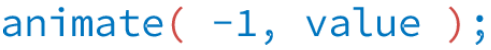
The first argument is the *case* constant; the second argument is the input variable, *value*. For instance, for the statement `case 2`, your *animate* method call is



```
animate( 2, value );
```

Description

For the default case, the method call should be



```
animate( -1, value );
```

Description

To test your code, compile the *SelectionPractice2Controller.java* file and run the *SelectionPractice2Application* application. When the program begins, you will see an empty graphics window with a text box as shown in **Figure 5.26.**

Figure 5.26 The Input Box of the Application

Description

To execute your *switch* statement, enter an integer in the text box and press the “Test” button. Depending on how you coded the *case* statements, the *break* statements, and the input value, the window will display (in green) the path of execution of your code. For example, **Figure 5.27** demonstrates the code path when the input value is 3. If the path is not what you expected, you will need to correct your code.

Figure 5.27 A Sample Run of the Application

Description

To test your code, enter each integer from 0 to 4, plus some other integer value into the text box, pressing the “Test” button each time. To exit the application, close the window.

DISCUSSION QUESTIONS

1. Explain the purpose of the *default* case in a *switch* statement.
2. Explain what happens when you omit the *default* case.

© Lois GoBe/Shutterstock.

CHAPTER REVIEW

5.13 Chapter Summary

- Java provides equality, relational, and logical operators to evaluate a condition, and selection statements to choose which instructions to execute based on whether a condition evaluates to *true* or *false*.
- The equality operators (`==`, `!=`) are used to test whether two operands are equal. The operands are expressions that evaluate to a primitive numeric or *boolean* type or an object reference.
- The relational operators (`<`, `<=`, `>`, `>=`) compare the values of two operands that are expressions that evaluate to a primitive numeric type.
- The logical operators (`!`, `&&`, and `||`) take *boolean* expressions as operands. The logical NOT (`!`) takes one operand, and inverts its value, changing *true* to *false* and *false* to *true*. The AND operator (`&&`) takes two *boolean* expressions as operands; if both operands are *true*, then the result is *true*; otherwise, the result is *false*. The OR operator (`||`) also takes two *boolean* expressions as operands. If both operands are *false*, then the result is *false*; otherwise, the result is *true*.
- The logical NOT operator (`!`) has the highest precedence of these operators, followed by the relational operators,

then the equality operators, then the logical AND (`&&`), then the logical OR(`||`).

- DeMorgan's Laws can be used to form equivalent logical expressions to improve readability of the code.
- The *if* statement is used to perform certain operations for one set of data and to do nothing for all other data.
- Curly braces are required when the true or false block of an *if* statement consists of more than one statement.
- The *if/else* statement is used to perform certain operations for one set of data and other operations for all other data.
- The *if/else if* statement is appropriate when the data falls into more than two mutually exclusive categories and the appropriate instructions to execute are different for each category.
- *if/else* statements can be coded sequentially and can be nested inside other *if/else* statements.
- When *if* statements are nested, the compiler matches any *else* clause with the most previous *if* condition that doesn't already have an *else* clause.
- Because rounding errors can be introduced in floating-point calculations, do not use the equality operators to compare two floating-point numbers. Instead, compare the absolute value of the difference between the numbers to some threshold value.
- When we need exact precision in calculations with decimal numbers, you can use the *BigDecimal* class in the Java Class Library.
- Using the equality operator on object references compares the values of the references, not the object data. Two object references will be equal only if they point to the same object.
- Use the *equals* method to determine whether the data in two objects are equal.
- In addition to the *equals* method, two *Strings* can also be compared using the *equalsIgnoreCase* method and the *compareTo* method of the *String* class.
- The conditional operator (?:) is used in expressions where

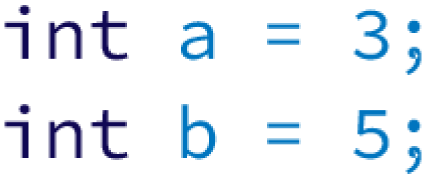
one of two values should be used depending on the evaluation of a condition. The conditional operator is useful for validating input and for outputting similar messages.

- The *switch* statement evaluates an integer or character expression or a *String*, then compares the expression's value to *case* constants. When a match is found, it executes the corresponding statement or block and exits the *switch* block.

5.14 Exercises, Problems, and Projects

5.14.1 Multiple Choice Exercises

1. Given the following code declaring and initializing two *int* variables *a* and *b* with respective values 3 and 5, indicate whether the value of each expression is *true* or *false*.



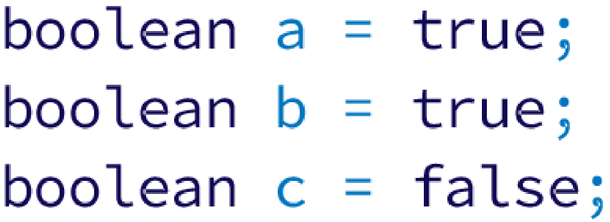
```
int a = 3;
int b = 5;
```

Description

Expression	true	false
• `a < b`	____	____
• `a != b`	____	____
• `a == 4`	____	____
• `( b - a ) <= 1`	____	____
• `Math.abs( a - b ) >= 2`	____	____
• `( b % 2 == 1 )`	____	____
• `b <= 5`	____	____

---

9. Given the following code declaring and initializing three *boolean* variables *a*, *b*, and *c*, with respective values *true*, *true*, and *false*, indicate whether the value of each expression is *true* or *false*.



```
boolean a = true;
boolean b = true;
boolean c = false;
```

Description

Expression	true	false
`!a`	____	____
`a && b`	____	____
`a && c`	____	____
`a || c`	____	____
`!( a || b )`	____	____
`!a || b`	____	____
`!( !( a && c ) )`	____	____
`a && !( b || c )`	____	____

18. Given two *boolean* variables *a* and *b*, are the following expressions equivalent?

    - `!( !a )`
    - `a`

21. Given two *boolean* variables *a* and *b*, are the following expressions equivalent?

    - `!( a && b )`
    - `!a || !b`

24. Given two *boolean* variables *a* and *b*, are the following expressions equivalent?

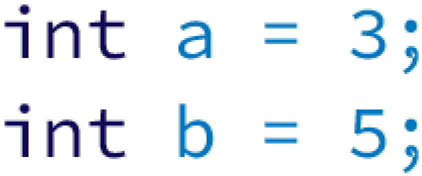
    - `!( !a && !b )`
    - `a && b`

27. Given two *boolean* variables *a* and *b*, are the following

expressions equivalent?

- `!( !a && !b )`
- `a || b`

30. Given the following code declaring and initializing two *int* variables *a* and *b* with respective values 3 and 5, indicate whether the operand (`b < 10`) will be evaluated.



```
int a = 3;
int b = 5;
```

Description

Expression	yes	no
• `a < b || b < 10`	____	____
• `a != b && b < 10`	____	____
• `a == 4 || b < 10`	____	____
• `a > b && b < 10`	____	____

---

35. Mark all the valid Java selection keywords.

- `if`
- `else if`
- `else`
- `elsif`

40. How do we compare the value of two *String* objects in Java? (Mark all that apply.)

- using the = operator
- using the == operator
- using the *equals* method

5.14.2 Reading and Understanding Code

1. What is the output of this code sequence?

   ```
   boolean a = true;
   System.out.println( a );
   ```

   Description

2. What is the output of this code sequence?

   ```
   boolean a = ( true && false );
   System.out.println( a );
   ```

   Description

3. What is the output of this code sequence?

   ```
   if ( ( true || false ) && ( false || true ) )
      System.out.println( "Inside true block" );
   System.out.println( "End of sequence" );
   ```

   Description

4. What is the output of this code sequence?

   ```
   if ( 27 % 3 == 0 )
      System.out.println( "27 is divisible by 3" );
   else
      System.out.println( "27 is not divisible by 3" );
   System.out.println( "End of sequence" );
   ```

   Description

5. What is the output of this code sequence?

   ```
   String s = "Hello";
   if ( s.equals( "hello" ) )
      System.out.println( "String is hello" );
   else
      System.out.println( "String is not hello" );
   System.out.println( "End of sequence" );
   ```

   Description

6. What is the output of this code sequence?

```
int grade = 77;
if ( grade >= 90 )
   System.out.println( "A" );
else if ( grade >= 80 )
   System.out.println( "B" );
else if ( grade >= 70 )
   System.out.println( "C" );
else
   System.out.println( "D or lower" );
System.out.println( "Done" );
```

   Description

7. What is the output of this code sequence?

```
int a = 65;
boolean b = false;

if ( a >= 70 )
{
   System.out.println( "Hello 1" );
   if ( b == true )
      System.out.println( "Hello 2" );
}
else
{
   System.out.println( "Hello 3" );
   if ( b == false )
      System.out.println( "Hello 4" );
}
System.out.println( "Done" );
```

   Description

8. Replace the block at lines 45–50 of Example 5.14 with a single line of code using the ?: operator.

9. What is the output of this code sequence?

```
int season = 3;
switch ( season )
{
   case 1->
        System.out.println( "Season is Winter" );
   case 2->
        System.out.println( "Season is Spring" );
   case 3->
        System.out.println( "Season is Summer" );
   case 4->
        System.out.println( "Season is Fall" );
   default->
        System.out.println( "Invalid season" );
}
```

Description

10. What is the output of this code sequence?

```
char c = 'e';
switch ( c )
{
   case 'H'->
        System.out.println( "letter 1" );
   case 'e'->
        System.out.println( "letter 2" );
   case 'l'->
        System.out.println( "letters 3 and 4" );
   case 'o'->
        System.out.println( "letter 5" );
   default->
        System.out.println( "letter is not in Hello" );
}
```

Description

5.14.3 Fill In the Code

For Exercises 20 through 30, assume that a *boolean* variable named *a* has been declared and assigned the value *true* or *false*. You should also assume that two *int* variables named *b* and *c* have been declared and assigned some integer values.

1. If *a* is *true*, increment *b* by 1.

```
// your code goes here
```

2. If *a* is *true*, increment *b* by 2; if *a* is *false*, decrement *b* by 1.

```
// your code goes here
```

3. If *a* is *true*, change *a* to *false*; if *a* is *false*, change *a* to *true*.

```
// your code goes here
```

4. If *b* is equal to *c*, then assign *true* to *a*.

```
// your code goes here
```

5. If *b* is less than *c*, increment *b* by 1; otherwise, leave *b* unchanged.

```
// your code goes here
```

6. If *b* is *a* multiple of *c*, set *a* to *true*; otherwise, set *a* to *false*.

```
// your code goes here
```

7. If *c* is not equal to 0, assign to *b* the value of *b* divided by *c*.

```
// your code goes here
```

8. If the product *b* times c is greater than or equal to 100, then invert *a* (if *a* is *true*, *a* becomes *false*; if *a* is *false*, *a* becomes *true*); otherwise, assign *true* to *a*.

```
// your code goes here
```

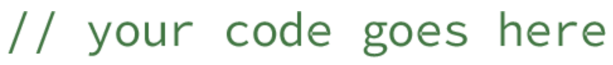
9. If *a* is *true* and *b* is greater than 10, increment *c* by 1.

```
// your code goes here
```

10. If both *b* and *c* are less than 10, then assign *true* to *a*; otherwise, assign *false* to *a*.

```
// your code goes here
```

11. If *b* or *c* is greater than 5, then assign *true* to *a*; otherwise, assign *false* to *a*.

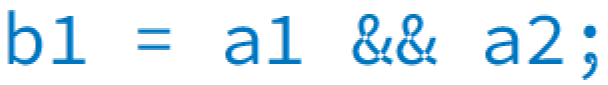


```
// your code goes here
```

5.14.4 Identifying Errors in Code

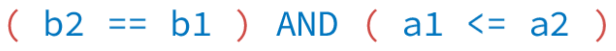
For Exercises 31 through 38, assume that two *boolean* variables named *b1* and *b2* have been declared and assigned the value *true* or *false* earlier in the program. You should also assume that two *int* variables named *a1* and *a2* have been declared and assigned some integer values earlier in the program.

1. Where is the error in this code sequence?



```
b1 = a1 && a2;
```

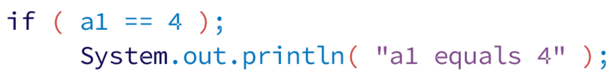
2. Where is the error in this expression?



```
( b2 == b1 ) AND ( a1 <= a2 )
```

Description

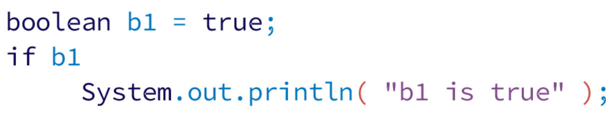
3. Where is the logical error in this code sequence?



```
if ( a1 == 4 );
      System.out.println( "a1 equals 4" );
```

Description

4. Where is the error in this code sequence?



```
boolean b1 = true;
if b1
      System.out.println( "b1 is true" );
```

Description

5. Where is the error in this code sequence?

```
if { b2 == true }
      System.out.println( "b2 is true" );
```

Description

6. Where is the error in this code sequence?

```
if ( b1 == true )
   System.out.println( "b1 is true" );
else
   System.out.println( "b1 is false" );
else if ( a1 < 100 )
   System.out.println( "a1 is <= 100" );
```

Description

7. Is there an error in this code sequence? Explain.

```
if ( b2 == b1 )
      System.out.println( "b2 and b1 have the same value" );
else if ( a1 == a2 )
      System.out.println( "a1 and a2 have the same value" );
else
      System.out.println( "All variables are different" );
```

Description

8. Is there an error in this code sequence? Explain.

```
if ( b2 )
      System.out.println( "b2 is true" );
else if ( a1 <= 10 || a2 > 50 )
{
      System.out.print( "a1 <= 10 or " );
      System.out.println( "a2 > 50" );
}
else
      System.out.println( "none of the above" );
```

Description

5.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. You coded the following in class *Test.java:*

```
boolean b = true;
if ( b )
      System.out.println( "Inside true block" );
      System.out.println( "b was true" );
else         // line 12
      System.out.println( "Inside false block" );
```

Description

At compile time, you get the following error:

```
Test.java:12: error: 'else' without 'if'
else      // line 12
^
1 error
```

Description

Explain what the problem is and how to fix it.

2. You coded the following in the class *Test.java:*

```
int a = 32;
if ( a = 31 )     // line 9
      System.out.println( "The value of a is 31" );
else
      System.out.println( "The value of a is not 31" );
```

Description

At compile time, you get the following error:

```
Test.java:9: error: incompatible types
if ( a = 31 )    // line 9
       ^
1 error
```

Description

Explain what the problem is and how to fix it.

3. You coded the following in the class *Test.java*:

```
boolean b = true;
if ( b )
{
      System.out.println( "Inside true block" );
      System.out.println( "b was true" );
else         // line 13
      System.out.println( "Inside false block" );
}
System.out.println( "Done" );
```

Description

At compile time, you get the following error:

```
Test.java:13: error: 'else' without 'if'.
else         // line 13
^
1 error
```

Description

Explain what the problem is and how to fix it.

5.14.6 Write a Short Program

1. Write a program that takes two *ints* as input from the keyboard, representing the number of hits and the number of at-bats for a batter. Then calculate the batter's hitting percentage and check if the hitting percentage is above .300. If it is, output that the player is eligible for the All Stars Game; otherwise, output that the player is not eligible.
2. Write a program that reads a *char* as an input from the keyboard and outputs whether it is a valid character to start an identifier. (Hint: look for a method in the *Character*

class.)

3. Write a program that calculates the area of the following figures:

    - a square of side 0.666666667
    - a rectangle of sides 1/9 and 4

    Test the two calculated areas for equality; discuss your result.

6. Write a program that reads a sentence from the keyboard. Depending on the last character of the sentence, print a message identifying the sentence as declarative (ends with a period), interrogative (ends with a question mark), exclamatory (ends with an exclamation point), or other.
7. An email address contains the @ character. Write a program that takes a word from the keyboard and outputs whether it is an email address based on the presence of the @ character. Do not worry about what else is in the word.

8. Write a program that takes two words as input from the keyboard, representing a password and the same password again. (Often, websites ask users to type their password twice when they register to make sure there was no typo the first time around.) Your program should do the following:

    - if both passwords match, then output "You are now registered as a new user"
    - otherwise, output "Sorry, there is a typo in your password"

11. Write a program that takes a word as input from the keyboard, representing a user ID. (Often, websites place constraints on user IDs.) Your program should do the following:

    - if the user ID contains between 6 and 10 characters inclusive, then output "Welcome, barbara" (assuming *barbara* is the user ID entered)
    - otherwise, output "Sorry, user ID invalid"

5.14.7 Programming Projects

1. Write a program that reads a web address (for instance, *www.yahoo.com*) from the keyboard and outputs whether this web address is for a government, a university, a business, an organization, or another entity.

    - If the web address ends with *gov*, it is a government web address.
    - If the web address ends with *edu*, it is a university web address.
    - If the web address ends with *com*, it is a business web address.
    - If the web address ends with *org*, it is an organization

web address.
    - Otherwise, it is a web address for another entity.

7. Write a program that reads a temperature as a whole number from the keyboard and outputs a “probable” season (winter, spring, summer, or fall) depending on the temperature.

    - If the temperature is greater than or equal to 90, it is probably summer.
    - If the temperature is greater than or equal to 70 and less than 90, it is probably spring.
    - If the temperature is greater than or equal to 50 and less than 70, it is probably fall.
    - If the temperature is less than 50, it is probably winter.
    - If the temperature is greater than 110 or less than –5, then you should output that the temperature entered is outside the valid range.

13. Write a program that takes a *String* as input from the keyboard, representing a year. Your program should do the following:

    - If the year entered has two characters, convert it to an *int*, add 2000 to it, and output it.
    - If the year entered has four characters, just convert it to an *int* and output it.
    - If the year entered has neither two nor four characters, output that the year is not valid.

17. Write a program that takes two words as input from the keyboard, representing a user ID and a password. Your program should do the following:

    - If the user ID and the password match “admin” and “open,” respectively, then output “Welcome.”
    - If the user ID matches “admin” and the password does

not match “open,” output “Wrong password.”

- If the password matches “open” and the user ID does not match “admin,” output “Wrong user ID.”
- Otherwise, output “Sorry, wrong ID and password.”

5.14.8 Technical Writing

1. When comparing two *doubles* or *floats* for equality, programmers calculate the difference between the two numbers and check if that difference is sufficiently small. Explain why and give a real-life example.
2. Look at the following code segment:

```
int b = 44;
if ( b = 23 )
      System.out.println( "Inside true block" );
```

Description

In Java, this code will generate the following compiler error:

```
Test.java:9: error: incompatible types
if ( b = 23 )
       ^
required: boolean
found: int
1 error
```

Description

In the C11 programming language, the equivalent code will compile and run and will give you the following output:

```
Inside true block
```

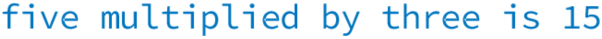
Discuss whether Java handles this situation better than C11 and why.

5.14.9 Group Project (for a group of 1, 2, or 3 students)

1. We want to build a simple “English language” calculator that does the following:

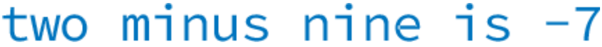
   - takes three inputs from the keyboard, two of them single digits (0 to 9)
   - takes a *char* from the keyboard, representing one of five operations from the keyboard: 1 (addition), – (subtraction), * (multiplication), / (division), and ^ (exponentiation)
   - outputs the description of the operation in plain English, as well as the numeric result

   For instance, if the two numbers are 5 and 3, and the operation is *, then the output should be



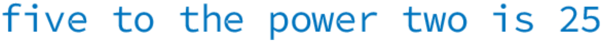
```
five multiplied by three is 15
```

   Note that the result is given as a number, not a word.
   If the two numbers are 2 and 9, and the operation is –, then the output should be



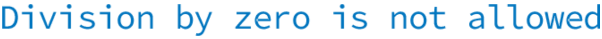
```
two minus nine is -7
```

   If the two numbers are 5 and 2, and the operation is ^, then the output should be



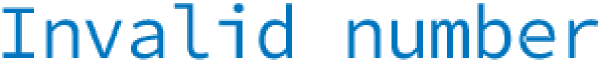
```
five to the power two is 25
```

   Hint: to perform the exponentiation, use the *pow* method of the *Math* class.
   If the two numbers are 5 and 0, and the operation is /, then the output should be



```
Division by zero is not allowed
```

   Here the operation will not be performed.
   If the two numbers are 25 and 3, and the operation is 1, then the output should be



```
Invalid number
```

because 25 has two digits.

As for the operators, they should be translated into English as follows:

+	plus
–	minus
*	multiplied by
/	divided by
^	to the power

You should use the *switch* … *case* selection statement to translate the input values into words.

You need to consider these special situations:

- For division, there is a special constraint: you cannot divide by 0, and you should therefore test whether the second number is 0. If it is 0, then you should output a message saying that you are not allowed to divide by 0.
- The “operator” is not one of the preceding five operators; in that case, output a message saying that the operator is not a valid one.
- One or two of the numbers is not a valid digit; again, you should output a message to that effect.
- Hint: You can deal with these special situations in the *default* statement of the *switch* block and possibly use some *boolean* variables to keep track of this information, as you may need it later in your program.

© Lois GoBe/Shutterstock.

CHAPTER 6

Flow of Control, Part 2: Looping

CHAPTER CONTENTS

Introduction
6.1 Event-Controlled Loops Using *while*
6.2 General Form for *while* Loops
6.3 Event-Controlled Looping

    6.3.1 Reading Data from the User
    6.3.2 Reading Data from a Text File

6.4 Looping Techniques

    6.4.1 Accumulation
    6.4.2 Counting Items
    6.4.3 Calculating an Average
    6.4.4 Finding Maximum or Minimum Values

6.5 Type-Safe Input Using *Scanner*
6.6 Constructing Loop Conditions
6.7 Testing Techniques for *while* Loops
6.8 Event-Controlled Loops Using *do/while*
6.9 **Programming Activity 1: Using *while* Loops**

6.10 Count-Controlled Loops Using *for*

- 6.10.1 Basic Structure of *for* Loops
- 6.10.2 Constructing *for* Loops
- 6.10.3 Testing Techniques for *for* Loops

6.11 Nested Loops
6.12 **Programming Activity 2: Using *for* Loops**
6.13 Chapter Summary
6.14 Exercises, Problems, and Projects

- 6.14.1 Multiple Choice Exercises
- 6.14.2 Reading and Understanding Code
- 6.14.3 Fill In the Code
- 6.14.4 Identifying Errors in Code
- 6.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
- 6.14.6 Write a Short Program
- 6.14.7 Programming Projects
- 6.14.8 Technical Writing
- 6.14.9 Group Project

Introduction

Have you ever watched the cashier at the grocery store? Let's call the cashier Jane. Jane's job is to determine the total cost of a grocery purchase. To begin, Jane starts with a total cost of $0.00. She then reaches for the first item and scans it to record its price, which is added to the total. Then she reaches for the second item, scans that item to record its price, which is added to the total, and so on. Jane continues scanning each item, one at a time, until there are no more items to scan. Usually, the end of an order is signaled by a divider bar lying across the conveyor belt. When Jane sees the divider bar, she knows she is finished. At that point, she tells us the total cost of the order, collects the money, and gives us a receipt.

So we see that Jane's job consists of performing some preliminary work, processing each item one at a time, and reporting the result at the end.

In computing, we often perform tasks that follow this same pattern:

1: initialize values
2: process items one at a time
3: report results

The flow of control that programmers use to complete jobs with this pattern is called **looping**, or **repetition**.

6.1 Event-Controlled Loops Using *while*

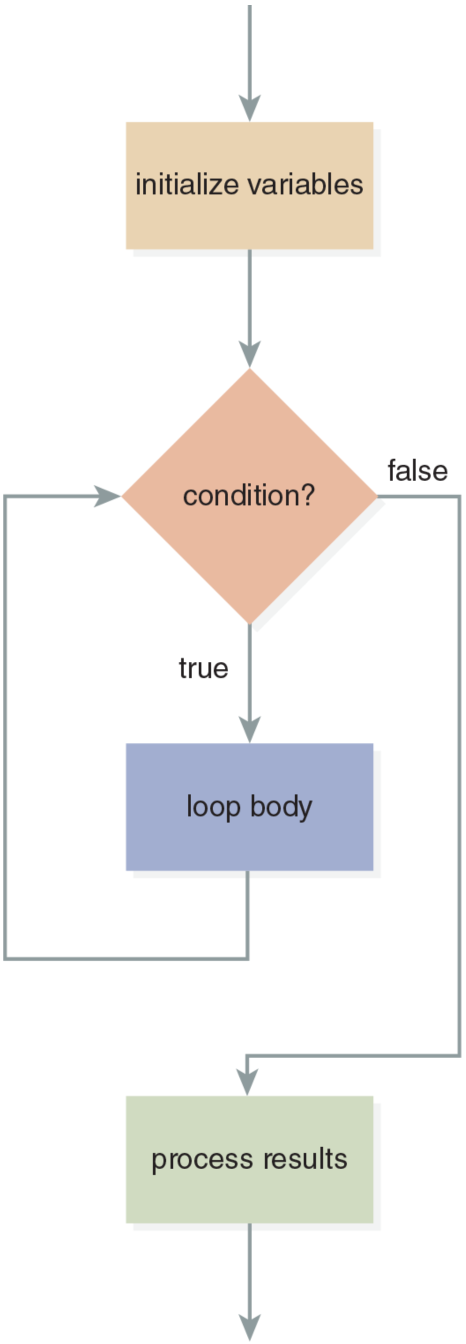
If we attempt to write pseudocode for the grocery store cashier, we may start with something like this:

```
set total to $0.00
reach for first item
if item is not the divider bar
   add price to total
reach for next item
if item is not the divider bar
   add price to total
reach for next item
if item is not the divider bar
   add price to total
… (finally)
reach for next item
item is the divider bar,
   tell the customer the total price
```

Description

We can see a pattern here. We start with an order total of $0.00. Then we repeat a set of operations for each item. We reach for the item and check whether it’s the divider bar. If the item is not the divider bar, we add the item’s price to the order total. We reach for the next item and check whether it’s the divider bar, and so on. When we reach for the item and find that it is the divider bar, we know there are no more items to process, so the total we have at that time is the total for the whole order. In other words, we don’t know the number of items that will be placed on the conveyor belt. We just process the order, item by item, until we see the divider bar, which we do not process.

In Java, the *while* loop is designed for repeating a set of instructions for each input value when we don't know at the beginning how many input values there will be. We simply process each input value, one at a time, until a signal—an event—tells us that there is no more input. This is called **event-controlled looping**. In the cashier's case, the signal for the end of input was the divider bar. In other tasks, the signal for the end of the input may be a special value that the user enters, called a **sentinel value**, or it may be that we've reached the end of an input file.


initialize variables
condition?
false
true
loop body
process results

6.2 General Form for *while* Loops

The while loop has this syntax:

```
// initialize variables
while ( condition )
{
   // process data; loop body
}
// process the results
```

Description

The condition is a *boolean* expression, that is, any expression that evaluates to *true* or *false*. When the *while* loop statement is encountered, the condition is evaluated; if the value is *true*, the statements in the **loop body** are executed. The condition is then reevaluated and, if *true*, the loop body is executed again. This repetition continues until the loop condition evaluates to *false*, at which time, the loop body is skipped and execution continues at the instruction following the loop body.

The curly braces are needed only if the loop body has more than one statement—that is, if more than one statement should be executed if the condition evaluates to *true*.

The scope of any variable defined within the *while* loop body extends from its declaration to the end of the *while* loop. Thus, any variable that is declared within a *while* loop body has block scope and cannot be referenced after the *while* loop ends.

The flow of control of a *while* loop is shown in **Figure 6.1**.

Figure 6.1 Flow of Control of a *while* Loop

Description

Each execution of the loop body is called an **iteration** of the loop. Thus, if the loop body executes five times before the condition evaluates to *false*, we say there were five iterations of the *while* loop.

What happens if the loop condition is *false* the first time it is evaluated? Because the loop condition is evaluated before executing the *while* loop body, and the loop body is executed only if the condition is *true*, it is possible that the *while* loop body is never executed. In that case, there would be **zero iterations** of the loop.

Using a *while* loop construct, the pseudocode for the cashier would look like this:

```
set total to $0.00
reach for first item
while item is not the divider bar
{
   add price to total
   reach for next item
}
// if we get here, the item is the divider bar
output the total price
```

Description

It is also possible to construct a *while* loop whose condition *never* evaluates to *false*. That results in an **endless loop**, also known as an **infinite loop**. Because the condition always evaluates to *true*, the loop body is executed repeatedly, without end. One symptom of an endless loop is that the program doesn't terminate; it appears to "hang." However, if the program writes some output in the loop body, we will see that output spewing out on the Java console. Normally, the only recourse is for the user to abort the program.

COMMON ERROR TRAP

Avoid putting a semicolon after the condition of a *while* loop. Doing so creates an empty loop body and could result in an endless loop.

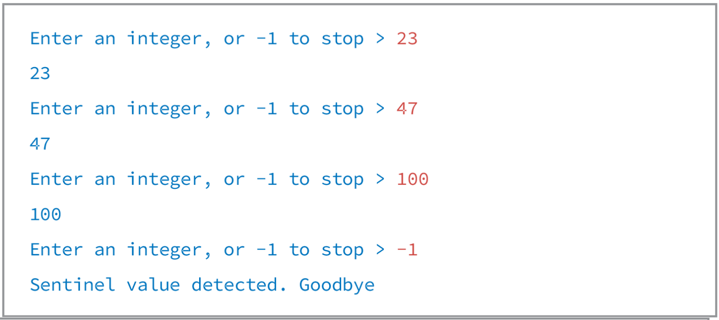
The way to ensure that the condition will eventually evaluate to *false* is to include code, called a **loop update statement**, within the loop body that appropriately changes the variable that is being tested by the loop condition. If, for example, the loop condition tests for reading a specific value, the loop update statement should read the next input value.

One common logic error that causes an endless loop is putting a semicolon after the condition, as in the following:

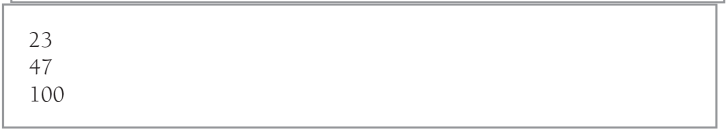
```
while ( condition ); // semicolon causes endless loop if condition is true
```

Description

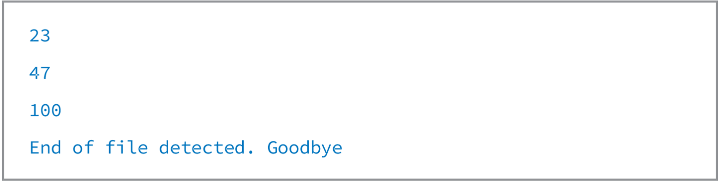
A semicolon immediately following the condition indicates an empty loop body. Although some advanced programming techniques call for the use of an empty loop body, we will not be using those techniques in this chapter.



```
Enter an integer, or -1 to stop > 23
23
Enter an integer, or -1 to stop > 47
47
Enter an integer, or -1 to stop > 100
100
Enter an integer, or -1 to stop > -1
Sentinel value detected. Goodbye
```



```
23
47
100
```



```
23
47
100
End of file detected. Goodbye
```

6.3 Event-Controlled Looping

The *while* loop is used when we don’t know how many times the loop will execute; that is, when the loop begins, we don’t know how many iterations of the loop will be required. We rely on a signal, or **event**, to tell us that we have processed all the data. For example, when the cashier begins checking out an order, she doesn’t (necessarily) know how many items are in the grocery cart; she only knows to stop when she sees the divider bar on the conveyor belt. We call this an event-controlled loop because we continue processing data until an event occurs, which signals the end of the data.

When we’re prompting the user to enter data from the console, and we don’t know at the beginning of the loop how much data the user has to be processed, we can define a special input value, called the sentinel value. The sentinel value can vary from task to task and is typically a value that is outside the normal range of data for that task.

Sometimes the data our program needs are in a text file. For example, a file could store a company’s monthly sales for the last five years. We may want to calculate average monthly sales or perform other statistical computations on that data. In this case, we need to read our data from the file, instead of asking the user to enter the data from the keyboard. Typically, we use a file when a large amount of data is involved because it would be impractical for a user to enter the data manually.

Reading from a file is also an event-controlled loop because we don't know at the beginning of the program how much data is in the file. Thus, we need some way to determine when we have finished processing all the data in the file. Java enables us to test if we have reached the end of the file. Thus, for input from a file, sensing the end-of-file indication is the event that signals that there is no more data to read.

6.3.1 Reading Data from the User

Let's look at the general form for using a *while* loop to process data entered from the user.

```
initialize variables
read the first data item // priming read
while data item is not the sentinel value
{
    process the data

    read the next data item // update read
}
report the results
```

Description

After performing any initialization, we attempt to read the first item. We call this the **priming read** because, like priming a pump, we use that value to feed the condition of the *while* loop for the first iteration. If the first item is not the sentinel value, we process it. Processing may consist of calculating a total, counting the number of data items, comparing the data to previously read values, or any number of operations. Then we read the next data item. This is called the **update read** because we update the data item in preparation for feeding its value into the condition of the *while* loop for the next iteration. This processing, followed by an update read, continues until we do read the sentinel value, at which time we do not execute the *while* loop body. Instead, we skip to the first instruction following the *while* loop. Note that the sentinel value is not meant to be processed. Like the divider bar for the cashier, it is simply a signal to stop processing.

We illustrate this pattern in **Example 6.1**, which prompts the user for integers and echoes to the console whatever the user enters. We chose the sentinel value to be –1; that is, when the user enters a –1, we stop processing.

Example 6.1 Echoing Input from the User

```java
/*  Working with a sentinel value
    Anderson, Franceschi
*/
import java.util.Scanner;

public class EchoUserInput
{
   public static void main( String [ ] args )
   {
      final int SENTINEL = -1;
      int number;

      Scanner scan = new Scanner( System.in );

      // priming read
      System.out.print( "Enter an integer, or -1 to stop > " );
      number = scan.nextInt( );

      while ( number != SENTINEL )
      {
          // processing
          System.out.println( number );

          // update read
          System.out.print( "Enter an integer, or -1 to stop > " );
          number = scan.nextInt( );
      }

      System.out.println( "Sentinel value detected. Goodbye" );
   }
}
```

Description

Figure 6.2 shows the output from this program when the user enters 23, 47, 100, and –1.

Figure 6.2 Output from Example 6.1, Using a Sentinel Value

Description

On line 10, we declare the sentinel value, −1, as a constant because the value of the sentinel will not change during the execution of the program, and it lets us clearly state via the *while* loop condition (line 19) that we want to execute the loop body only if the input is not the sentinel value.

Then on lines 16–17, we perform the priming read. The *while* loop condition on line 19 checks for the sentinel value. If the user enters the sentinel value first, we skip the *while* loop altogether and execute line 29, which prints a message that the sentinel value was entered, and we exit the program. If the user enters a number other than the sentinel value, we execute the body of the *while* loop (lines 21–26). In the *while* loop, we simply echo the user's input to the console, then perform the update read. Control then skips to the *while* loop condition, where the value the user entered in the update read is compared to the sentinel value. If this entry is the sentinel value, the loop is skipped; otherwise, the body of the loop is executed: The value is echoed, then a new value is read. This same processing continues until the user does enter the sentinel value. Note that the sentinel value (−1) is not echoed to the output.

COMMON ERROR TRAP

Omitting the update read may result in an endless loop.

A common error in constructing *while* loops is forgetting the update read. Without the update read, the *while* loop continually processes the same data item, leading to an endless loop.

Another common error is omitting the priming read and, instead, reading data inside the *while* loop before the processing, as in the following pseudocode:

```
initialize variables
while data item is not the sentinel value
{
     read the next data
     process the data
}
report the results
```

Description

This structure has several problems. The first time we evaluate the *while* loop condition, we haven't read any data, so the result of that evaluation is unpredictable. Second, when we do read the sentinel value, we will process it, leading to incorrect results.

COMMON ERROR TRAP

Omitting the priming read leads to incorrect results.

6.3.2 Reading Data from a Text File

The *Scanner* class enables us to read data easily from a text file. Java also provides a whole set of classes in the *java.io* package to enable programmers to perform user input and output with a file.

For the *Scanner* class, the general form for reading data from a text file is a little different from reading the data from the user. First, instead of reading a value and checking whether it is the sentinel value, we check whether there is more data in the file, then read a value. Second, we don't need to print a prompt because the user doesn't enter the data; we just read the next value from the file. With the *Scanner* class, the pseudocode for reading from a text file is shown here:

```
initialize variables
while we have not reached end of file
{
   read the next data item
   process the data
}
report the results
```

Description

Scanner class methods, including a constructor for reading from a text file, are shown in **Table 6.1**. Another class we will use is the *File* class, which associates a file name with a file.

The constructor for the *File* class is shown in **Table 6.2**.

Selected Methods of the *Scanner* Class

Selected Methods of the *Scanner* Class

Constructor

`Scanner( File file )`
creates a *Scanner* object and associates it with a file

Return value	Method name and argument list
`boolean`	`hasNext( )` returns *true* if there is another token in the input stream; *false*, otherwise
`byte`	`nextByte( )` returns the next input as a *byte*
`short`	`nextShort( )` returns the next input as a *short*
`int`	`nextInt( )` returns the next input as an *int*
`long`	`nextLong( )` returns the next input as a *long*
`float`	`nextFloat( )` returns the next input as a *float*
`double`	`nextDouble( )` returns the next input as a *double*
`boolean`	`nextBoolean( )` returns the next input as a *boolean*
`String`	`next( )` returns the next token in the input line as a *String*

Description

***File* Class Constructor**

A Constructor for the *File* Class

`File( String pathname )`

constructs a *File* object with the *pathname* file name so that the file name is platform independent

---

Description

The constructor shown in Table 6.1 can be used to associate a *Scanner* object with a file. The *Scanner* object will tokenize the contents of the file and return the tokens as we call the next methods. The *hasNext* method in the *Scanner* class returns *true* if the input has another token, and *false* otherwise. Thus, when the *hasNext* method returns *false*, we know we have reached the end of the file.

Example 6.2 reads integers from a file named *input.txt* and echoes the integers to the console. The contents of *input.txt* are shown in **Figure 6.3** and the output from the program is shown in **Figure 6.4**.

Example 6.2 Echoing Input from a File

```
 1 /* Reading a Text File
 2    Anderson, Franceschi
 3 */
 4 import java.util.Scanner;
 5 import java.io.File;
 6 import java.io.IOException;
 7
 8 public class EchoFileData
 9 {
10    public static void main( String [ ] args ) throws IOException
11    {
12       int number;
13
14       File inputFile = new File( "input.txt" );
15       Scanner file = new Scanner( inputFile );
16
17       while ( file.hasNext( ) )
18       {
19           // read next integer
20           number = file.nextInt( );
21           // process the value read
22           System.out.println( number );
23       }
24
25       System.out.println( "End of file detected. Goodbye" );
26    }
27 }
```

Description

Figure 6.3 Contents of *input.txt*

Description

Figure 6.4 Output from Example 6.2, Reading from a File

Description

On line 14 of Example 6.2, we use the constructor of the *File* class to convert the file name, *input.txt*, to a platform-independent file name. Because we are specifying the simple file name, the JVM will look for the file in the same folder as our source file. If the file is located in another folder, we need to specify the path as well as the file name. For example, if the file were located on a flash drive in a Windows system, we would pass the *String “e:\\input.txt”* to the constructor. Notice that we need to use an escape sequence of two backslashes in order to specify the pathname, *e:\input.txt*.

The *File* class belongs to the *java.io* package, so we include an *import* statement for that class in line 5.

In line 15, we construct a *Scanner* object associated with the *inputFile* object. If the file is not found, the constructor generates a *FileNotFoundException*. It is also possible that an *IOException* may be generated if we encounter problems reading the file. Java requires us to acknowledge that these exceptions may be generated. One way to do that is to include the phrase *throws IOException* in the header for *main* (line 10). We also import the *IOException* class on line 6.

On line 17, the first time our *while* loop condition is evaluated, we check whether there is any data in the file. If the file is empty, the *hasNext* method will return *false*, and we will skip execution of the loop body, continuing at line 25, where we print a message and exit the program.

REFERENCE POINT

You can read more about the *Scanner* class on Oracle's Java website: *www.oracle.com/technetwork/java*.

The body of the *while* loop (lines 19–22) calls the *nextInt* method to read the next integer in the file and echoes that integer to the console. We then reevaluate the *while* loop condition (line 17) to determine if more data is in the file. When no more integers remain to be read, the *hasNext* method returns *false*, and we skip to line 25, where we print a message and exit the program.

Notice that we do not use a priming read because the *hasNext* method essentially peeks ahead into the file to see if there is more data. If the *hasNext* method returns *true*, we know that there is another integer to read, so we perform the read in the first line of the *while* loop body (line 20).

```
Welcome to the addition calculator.

Enter the first number or 0 for the total > 34
Enter the next number or 0 for the total > -10
Enter the next number or 0 for the total > 2
Enter the next number or 0 for the total > 5
Enter the next number or 0 for the total > 8
Enter the next number or 0 for the total > 0
The total is 39
```

```
This program counts the number of passing test scores.
Enter a -1 to stop.
Enter the first score > 98
Enter the next score > 75
Enter the next score > 60
Enter the next score > 59
Enter the next score > 45
Enter the next score > 88
Enter the next score > 94
Enter the next score > 96
Enter the next score > 56
Enter the next score > 77
Enter the next score > 82
Enter the next score > 89
Enter the next score > 100
Enter the next score > 78
Enter the next score > 55
Enter the next score > -1
You entered 15 scores
The number of passing test scores is 11
73.3% of the class passed the test
```

```
To calculate a class average,
enter each test score.
When you are finished, enter a -1
Enter the first test score > 88
Enter the next test score > 78
Enter the next test score > 96
Enter the next test score > 75
Enter the next test score > 99
Enter the next test score > 56
Enter the next test score > 78
Enter the next test score > 84
Enter the next test score > 93
Enter the next test score > 79
Enter the next test score > 90
Enter the next test score > 85
Enter the next test score > 79
Enter the next test score > 92
Enter the next test score > 99
Enter the next test score > 94
Enter the next test score > -1

The class average is 85.3
```

```
88
78
96
75
99
56
78
84
93
79
90
85
79
90
85
79
92
99
94
```

```
This program finds the maximum grade for a class
The maximum grade is 99
```

6.4 Looping Techniques

The *while* loop is an important tool for performing many common programming operations on a set of input values. For example, the *while* loop can be used to calculate the sum of values, count the number of values, find the average value, find the minimum and maximum values, and perform other operations.

6.4.1 Accumulation

Let's look at a common programming operation for which a *while* loop is useful: calculating the sum of a set of values. To do this, we will build a simple calculator that performs one function: addition. We will prompt the user for numbers one at a time. We'll make the sentinel value a 0; that is, when the user wants to stop, the user will enter a 0. At that point, we will print the total.

The calculator can be developed using an event-controlled *while* loop and a standard computing technique: **accumulation.** In the accumulation operation, we initialize a *total* variable to 0. Each time we input a new value, we add that value to the *total*. When we reach the end of the input, the current value of *total* is the total for all the input.

Here is the pseudocode for the addition calculator:

```
set total to 0
read a number // priming read
while the number is not the sentinel value
{
    add the number to total
    read the next number // update read
}
output the total
```

Description

Notice that this operation is almost identical to the grocery cashier's job in that we perform a priming read before the *while* loop. Inside the *while* loop, we process each number one at a time—adding each number to the total, then we read the next value, until we see the sentinel value, which is the signal to stop.

Example 6.3 provides the code for the addition calculator, and **Figure 6.5** shows the output for a sample execution of the calculator.

Example 6.3 An Addition Calculator

```
 1 /*  Addition Calculator
 2     Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6
 7 public class Calculator
 8 {
 9    public static void main( String [ ] args )
10    {
11       final int SENTINEL = 0;
12       int number;
13       int total = 0;
14
15       Scanner scan = new Scanner( System.in );
16
17       System.out.println( "Welcome to the addition calculator.\n" );
18
19       System.out.print( "Enter the first number"
20                         + " or 0 for the total > " );
21       number = scan.nextInt( );
22
23       while ( number != SENTINEL )
24       {
25           total += number;
26
27           System.out.print( "Enter the next number"
28                             + " or 0 for the total > " );
29           number = scan.nextInt( );
30       }
31
32       System.out.println( "The total is " + total );
33    }
34 }
```

Description

Figure 6.5 Output from a Sample Run of the Addition Calculator

Description

Line 13 declares and initializes the *total* to 0. This is an important step because the loop body will add each input value to the total. If the total is not set to 0 before the first input, we will get incorrect results. Furthermore, if *total* is declared but not initialized, our program will not compile.

COMMON ERROR TRAP

Forgetting to initialize the total to 0 will produce incorrect results.

Lines 19–21 read the first input value (the priming read). The *while* loop begins at line 23, and its condition checks for the sentinel value. The first time the *while* loop is encountered, this condition will check the value of the input from the priming read.

The loop body processes the input (line 25), which consists of adding the input value to the *total*. The final step in the loop body (lines 27–29) is to read the next input (the update read).

SOFTWARE ENGINEERING TIP

Indent the body of a *while* loop to clearly illustrate the logic of the program.

When the end of the loop body is reached, control is transferred back to line 23, where the loop condition is again tested with the input value read on line 29. If the condition is *true*, that is, if the input just read is not the sentinel value, then the loop body is reexecuted and the condition is retested, continuing until the input is the sentinel value, which causes the condition to evaluate to *false*. At that time, the loop body is skipped and line 32 is executed, which reports the results by printing the *total*.

COMMON ERROR TRAP

Choosing the wrong sentinel value may result in logic errors.

Notice that the body of the *while* loop is indented and that the opening and closing curly braces are aligned in the same column as the *w* in the *while*. This style lets us easily see which statements belong to the *while* loop body.

It is important to choose the sentinel value carefully. Obviously, the sentinel value cannot be a value that the user might want to be processed. In the addition calculator, we want to allow the user to enter positive or negative integers. We chose 0 as the sentinel value for two reasons. First, adding 0 to a total has no effect, so it is unlikely that the user will want to enter that value to be processed. Second, to the user, it is logical to enter a 0 to signal that there are no more integers to be added.

CODE IN ACTION

To see a step-by-step illustration of a *while* loop with a

sentinel value, look for the movie within the online resources. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

6.4.2 Counting Items

Counting is used when we need to know how many items are input or how many input values fit some criterion, for example, how many items are positive numbers or how many items are odd numbers. Counting is similar to accumulation in that we start with a count of 0 and increment (add 1 to) the count every time we read a value that meets the criterion. When there are no more values to read, the count variable contains the number of items that meet our criterion.

For example, let's count the number of students who passed a test. The pseudocode for this operation is as follows:

```
set countPassed to 0
read a test score
while the test score is not the sentinel value
{
   if the test score >= 60
   {
      add 1 to countPassed
   }
   read the next test score
}
output countPassed
```

Description

The application in **Example 6.4** counts the number of students that passed a test. We also calculate the percentage of the class that passed the test. To do this, we maintain a second count: the number of scores entered. This value will be incremented each time we read a score, whereas the *countPassed* value will be incremented only if the score is greater than or equal to 60. The sentinel value is –1. A sample run of this program is shown in Figure 6.6.

Example 6.4 Counting Passing Test Scores

```
 1 /* Counting passing test scores
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6 import java.text.DecimalFormat;
 7
 8 public class CountTestScores
 9 {
10    public static void main( String [ ] args )
11    {
12      int countPassed = 0;
13      int countScores = 0;
14      int score;
15      final int SENTINEL = -1;
16
17      Scanner scan = new Scanner( System.in );
18
19      System.out.println( "This program counts "
20                  + "the number of passing test scores." );
21      System.out.println( "Enter a -1 to stop." );
22
23      System.out.print( "Enter the first score > " );
24      score = scan.nextInt( );
25
26      while ( score != SENTINEL )
27      {
28          if ( score >= 60 )
29          {
30                countPassed++;
31          }
32
33          countScores++;
34
35          System.out.print( "Enter the next score > " );
36          score = scan.nextInt( );
37      }
38
39      System.out.println( "You entered " + countScores + " scores" );
40      System.out.println( "The number of passing test scores is "
41                               + countPassed );
42      if ( countScores != 0 )
43      {
44         DecimalFormat percent = new DecimalFormat( "0.0%" );
45         System.out.println(
46           percent.format( (double)  countPassed  / countScores )
47           + " of the class passed the test" );
48      }
49    }
50 }
```

Description

Figure 6.6 Counting Passing Test Scores

Description

Lines 12 and 13 declare the variables *countPassed* and *countScores* and initialize both to 0. Initializing these values to 0 is critical; otherwise, we will get the wrong results or a compiler error. We initialize these values to 0 because at that point, we have not yet processed any test scores.

Our *while* loop framework follows the familiar pattern. We perform the priming read for the first input (lines 23–24); our *while* loop condition checks for the sentinel value (line 26); and the last statements of the *while* loop (lines 35–36) read the next value.

COMMON ERROR TRAP

Forgetting to initialize the count variables will produce a compiler error.

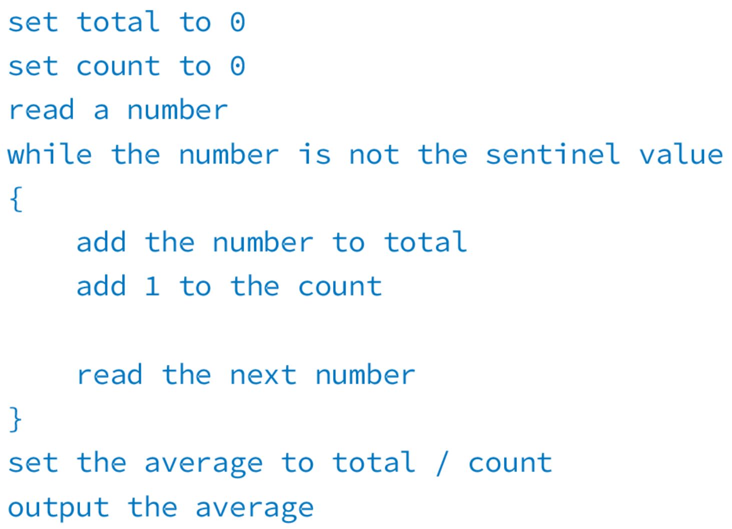
In the processing portion of the *while* loop, line 28 checks if the score just read is a passing score, and if so, line 30 adds 1 to *countPassed*. For each score entered, regardless of whether the student passed, we increment *countScores* (line 33).

When the sentinel value is entered, the *while* loop condition evaluates to *false* and control skips to line 39, where we output the number of scores entered and the number of passing scores. So that we avoid dividing by 0, note that line 42 checks whether no scores were entered. Note also that in line 46 we type cast *countPassed* to a *double* to force floating-point division, rather than integer division, so that the fractional part of the quotient will be maintained.

6.4.3 Calculating an Average

Calculating an average is a combination of accumulation and counting. We use accumulation to calculate the total and we use counting to count the number of items to average.

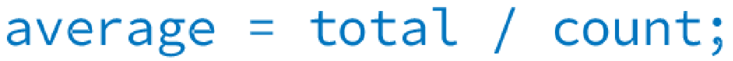
Here’s the pseudocode for calculating an average:



```
set total to 0
set count to 0
read a number
while the number is not the sentinel value
{
    add the number to total
    add 1 to the count

    read the next number
}
set the average to total / count
output the average
```

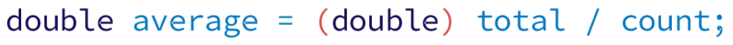
Description

Thus, to calculate an average test score for the class, we need to calculate the total of all the test scores, then divide by the number of students who took the test.



```
average = total / count;
```

It’s important to remember that if we declare *total* and *count* as integers, then the *average* will be calculated using integer division, which truncates the remainder. To get a floating-point average, we need to type cast one of the variables (either *total* or *count*) to a *double* or a *float* to force the division to be performed as floating-point.



```
double average = (double) total / count;
```

Description

The application in **Example 6.5** calculates an average test score for a class of students. The output is shown in **Figure 6.7**.

Example 6.5 Calculating an Average Test Score

```
 1 /* Calculate the average test score
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6 import java.text.DecimalFormat;
 7
 8 public class AverageTestScore
 9 {
10   public static void main( String [ ] args )
11   {
12     int count = 0;
13     int total = 0;
14     final int SENTINEL = -1;
15     int score;
16
17     Scanner scan = new Scanner( System.in );
18
19     System.out.println( "To calculate a class average," );
20     System.out.println( "enter each test score." );
21     System.out.println( "When you are finished, enter a -1" );
22
23     System.out.print( "Enter the first test score > " );
24     score = scan.nextInt( );
25
26     while ( score != SENTINEL )
27     {
28        total += score;   // add score to total
29        count++;          // add 1 to count of test scores
30
31        System.out.print( "Enter the next test score > " );
32        score = scan.nextInt( );
33     }
34
35     if ( count != 0 )
36     {
37         DecimalFormat oneDecimalPlace = new DecimalFormat( "0.0" );
38         System.out.println( "\nThe class average is "
39           + oneDecimalPlace.format( (double) total / count ) );
40     }
41     else
42         System.out.println( "\nNo grades were entered" );
43   }
44 }
```

Description

Figure 6.7 Calculating the Average Test Score

Description

In Example 6.5, lines 12 and 13 declare both *count* and *total* variables as *ints* and initialize each to 0. Again, our *while* loop structure follows the same pattern. Lines 23–24 read the first input value; the *while* loop condition (line 26) checks for the sentinel value; and the last statements in the *while* loop (lines 31–32) read the next score. For the processing portion of the *while* loop, we add the score to the total and increment the count of scores (lines 28–29). When the sentinel value is entered, we stop executing the *while* loop and skip to line 35.

In line 35, we avoid dividing by 0 by checking whether *count* is 0 (that is, if no scores were entered) before performing the division. If *count* is 0, we simply print a message saying that no grades were entered. If *count* is not 0, we calculate and print the average. We first instantiate a *DecimalFormat* object (line 37) so that we can output the average to one decimal place. Remember that we need to type cast the total to a *double* (lines 38–39) to force floating-point division, rather than integer division.

COMMON ERROR TRAP

Forgetting to check whether the denominator is 0 before performing division is a logic error.

6.4.4 Finding Maximum or Minimum Values

In previous examples, we calculated a total for a group of numbers by keeping a running total. We started with a total of 0, then added each new input value to the running total. Similarly, we counted the number of input items by keeping a running count. We started with a count of 0 and incremented the count each time we read a new value. We can apply that same logic to calculating a maximum or minimum. For example, to find the maximum of a group of values, we can keep a “running,” or current, maximum. We start by assuming that the first value we read is the maximum. In fact, it is the largest value we have seen so far. Then as we read each new value, we compare it to our current maximum. If the new value is greater, we make the new value our current maximum. When we come to the end of the input values, the current maximum is the maximum for all the input values.

Finding the minimum value, of course, uses the same approach, except that we replace the current minimum only if the new value is less than the current minimum.

Here’s the pseudocode for finding a maximum value in a file:

```
read a first number and make it the maximum
while there is another number to read
{
    read the next number
    if number > maximum
    {
       set maximum to number
    }
}
output the maximum
```

Description

Example 6.6 shows the code to find a maximum test grade in a file. As shown in **Figure 6.8**, the grades are stored as integers, one per line, in the file *grades.txt*. When this program runs, its output is shown in **Figure 6.9**.

Example 6.6 Finding the Maximum Value

```
 1 /* Find the maximum test grade
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6 import java.io.*;
 7
 8 public class FindMaximumGrade
 9 {
10    public static void main( String [ ] args ) throws IOException
11    {
12      int maxGrade;
13      int grade;
14
15      Scanner file = new Scanner( new File( "grades.txt" ) );
16
17      System.out.println( "This program finds the maximum grade "
18                          + "for a class" );
19
20      if ( ! file.hasNext( ) )
21      {
22           System.out.println( "No test grades are in the file" );
23      }
24      else
25      {
26          maxGrade = file.nextInt( ); // make first grade the max
27
28          while ( file.hasNext( ) )
29          {
30             grade = file.nextInt( ); // read next grade
31
32             if ( grade > maxGrade )
33               maxGrade = grade; // save as current max
34          }
35
36          System.out.println( "The maximum grade is " + maxGrade );
37      }
38   }
39 }
```

Description

Figure 6.8 The Contents of *grades.txt*

Description

Figure 6.9 Finding the Maximum Value

Description

In line 20, we call the *hasNext* method to test whether the file is empty. If so, we print a message (line 22) and the program ends. If, however, the file is not empty, we read the first value and automatically make it our maximum by storing the grade in *maxGrade* (line 26). In line 28, our *while* loop condition tests whether we have reached the end of the file. If not, we execute the body of the *while* loop (lines 30–33). We read the next grade and check whether that grade is greater than the current maximum. If so, we assign that grade to *maxGrade*; otherwise, we leave *maxGrade* unchanged. Then control is transferred to line 28 to retest the *while* loop condition.

When we do reach the end of the file, the *while* loop condition becomes *false*; control is transferred to line 36, and we output *maxGrade* as the maximum value.

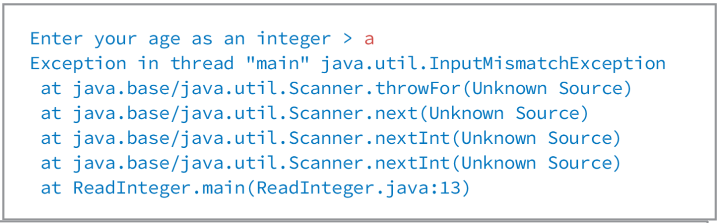
COMMON ERROR TRAP

Initializing a maximum or a minimum to an arbitrary value, such as 0 or 100, is a logic error and could result in incorrect results.

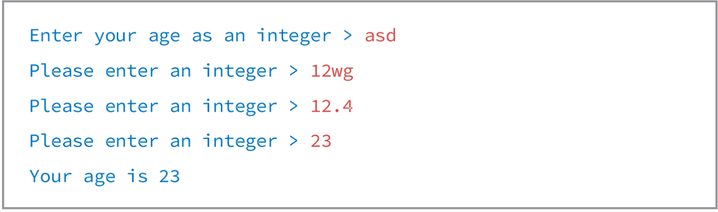
A common error is to initialize the maximum or minimum to an arbitrary value, such as 0 or 100. This will not work for all conditions, however. For example, let's say we are finding the maximum number and we initialize the maximum to 0. If the user enters all negative numbers, then when the end of data is encountered, the maximum will still be 0, which is clearly an error. The same principle is true when finding a minimum value. If we initialize the minimum to 0, and the user enters all positive numbers greater than 0, then at the end of our loop, our minimum value will still be 0, which is also incorrect.

with these end-of-chapter questions

6.14.1	Multiple Choice Exercises Question 1
6.14.2	Reading and Understanding Code Questions 5, 6, 7, 8, 20
6.14.3	Fill In the Code Questions 21, 22, 23, 24, 25, 26
6.14.4	Identifying Errors in Code Questions 30, 31
6.14.5	Debugging Area Question 37
6.14.6	Write a Short Program Questions 44, 45
6.14.8	Technical Writing Questions 70, 71



```
Enter your age as an integer > a
Exception in thread "main" java.util.InputMismatchException
 at java.base/java.util.Scanner.throwFor(Unknown Source)
 at java.base/java.util.Scanner.next(Unknown Source)
 at java.base/java.util.Scanner.nextInt(Unknown Source)
 at java.base/java.util.Scanner.nextInt(Unknown Source)
 at ReadInteger.main(ReadInteger.java:13)
```

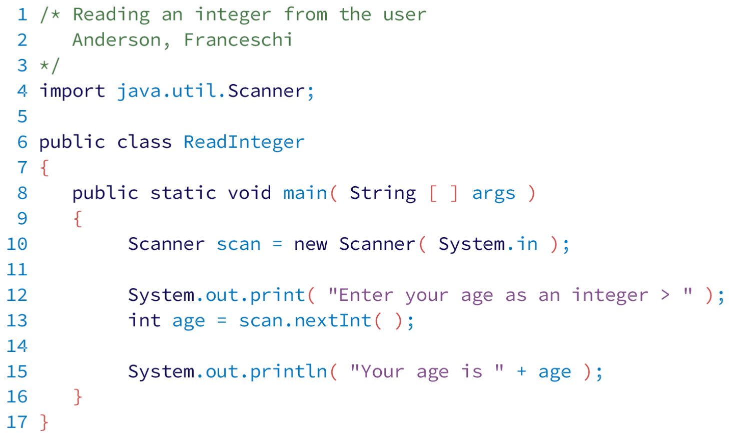


```
Enter your age as an integer > asd
Please enter an integer > 12wg
Please enter an integer > 12.4
Please enter an integer > 23
Your age is 23
```

6.5 Type-Safe Input Using *Scanner*

One problem with reading input using *Scanner* is that if the next token does not match the data type we expect, an *InputMismatchException* is generated, which stops execution of the program. This could be caused by a simple typo on the user's part; for example, the user may type a letter or other nonnumeric character when our program prompts for an integer. To illustrate this problem, **Example 6.7** shows a small program that prompts the user for an integer and calls the *nextInt* method of the *Scanner* class to read the integer, and **Figure 6.10** shows the *InputMismatchException* generated when the user enters an *a* instead of an integer. Notice that the program ends when the exception is generated; we never execute line 15, which would echo the age to the console.

Example 6.7 Reading an Integer



```java
/* Reading an integer from the user
   Anderson, Franceschi
*/
import java.util.Scanner;

public class ReadInteger
{
   public static void main( String [ ] args )
   {
        Scanner scan = new Scanner( System.in );

        System.out.print( "Enter your age as an integer > " );
        int age = scan.nextInt( );

        System.out.println( "Your age is " + age );
   }
}
```

Description

Figure 6.10 Input Failure

Description

We can make our program more robust by checking, before we read, that the next token matches our expected input. The *Scanner* class provides *hasNext* methods for doing this, which are shown in **Table 6.3**. The *hasNext* methods return *true* if the next token can be read as the data type requested. For example, if we expect an integer, we can test whether the user has typed characters that can be interpreted as an integer by calling the *hasNextInt* method. If that method returns *true*, it is safe to read the value using the *nextInt* method. If the next token is not what we need, that is, if the *hasNextInt* method returns *false*, then reading that value as an *int* will generate the *InputMismatchException*. In that case, we need to notify the user that the value typed is not valid and reprompt for new input. But first we need to clear the invalid input. We can flush the invalid input by calling the *nextLine* method of the *Scanner* class, which returns any remaining tokens on the input line as a *String*. Then we just ignore that *String*. **Example 6.8** shows a revised version of Example 6.7 that is type-safe, meaning we guarantee we have an integer to read before reading it.

Scanner Methods for Testing Tokens

Selected Input Stream Testing Methods of the *Scanner* Class

Return value	Method name and argument list
`boolean`	`hasNext( )` returns *true* if there is another token in the input stream; *false*, otherwise
`boolean`	`hasNextByte( )` returns *true* if the token in the input stream can be read as a *byte*; *false*, otherwise
`boolean`	`hasNextShort( )` returns *true* if the token in the input stream can be read as a *short*; *false*, otherwise
`boolean`	`hasNextInt( )` returns *true* if the token in the input stream can be read as an *int*; *false*, otherwise
`boolean`	`hasNextLong( )` returns *true* if the token in the input stream can be read as a *long*; *false*, otherwise
`boolean`	`hasNextFloat( )` returns *true* if the token in the input stream can be read as a *float*; *false*, otherwise
`boolean`	`hasNextDouble( )` returns *true* if the token in the input stream can be read as a *double*; *false*, otherwise
`boolean`	`hasNextBoolean( )` returns *true* if the token in the input stream can be read as a *boolean*; *false*, otherwise
`String`	`nextLine( )` returns the remainder of the input line as a *String*

Description

Example 6.8 Type-Safe Input

```
 1 /* Type-Safe Input Using Scanner
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6
 7 public class TypeSafeReadInteger
 8 {
 9    public static void main( String [ ] args )
10    {
11       Scanner scan = new Scanner( System.in );
12       String garbage;
13
14       System.out.print( "Enter your age as an integer > " );
15       while ( ! scan.hasNextInt( ) )
16       {
17             garbage = scan.nextLine( );
18             System.out.print( "\nPlease enter an integer > " );
19       }
20       int age = scan.nextInt( );
21       System.out.println( "Your age is " + age );
22    }
23 }
```

Description

On line 14 of Example 6.8, we prompt for the integer. Then on line 15, the *while* loop condition checks whether the user has, indeed, typed an integer value. If not, we ignore whatever the user did type by calling the *nextLine* method (line 17). On line 18, we reprompt the user. The *while* loop continues executing until the user does enter an integer and the *hasNextInt* method returns *true*. At that point, we execute line 20, which reads the integer into the *age* variable. Note that we need to use a *while* loop rather than an *if* statement because an *if* statement will test only the first invalid input value. Using a *while* loop allows us to keep prompting the user as many times as needed until a valid value is entered. **Figure 6.11** shows the output of this program when the user enters data other than integers, then finally enters an integer.

Figure 6.11 Reprompting Until the User Enters an Integer

Description

```
Welcome to Bonnie's Burgers. Select from our menu:
        C  Cheeseburger: $7.49
        H  Hot dog: $6.99
        L  Lemonade: $2.50
        T  Iced tea: $2.75
Order an item, or "S" to stop > h
Hot dog ordered. Subtotal: $6.99

Order another item, or "S" to stop > T
Iced tea ordered. Subtotal: $9.74

Order another item, or "S" to stop > s

Total order cost is $9.74
```

6.6 Constructing Loop Conditions

Constructing the correct loop condition may seem a little counterintuitive. The loop executes as long as the loop condition evaluates to *true*. Thus, if we want our loop to terminate when we read the sentinel value, then the loop condition should check that the input value is *not* the sentinel value. In other words, the loop continuation condition is the inverse of the loop termination condition. For a simple sentinel-controlled loop, the condition normally follows this pattern:

```
while ( inputValue != sentinel )
```

In fact, we can see that the loop conditions in many of the examples in this chapter use this form of *while* loop condition.

For some applications, there may be multiple sentinel values. For example, suppose we provide a menu for a user with each menu option being a single character. The user can repeatedly select options from the menu, with the sentinel value being *S* for stop. To allow case-insensitive input, we want to recognize the sentinel value as either *S* or *s*. To do this, we need a compound loop condition, that is, a loop condition that uses a logical AND (`&&`) or logical OR (`||`) operator.

Our first inclination might be to form the condition this way, which is **incorrect:**

```
while ( option != 'S' || option != 's' ) // INCORRECT
```

Description

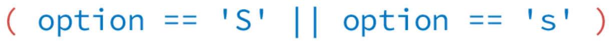
With this condition, the loop will execute forever. Regardless of what the user enters, the loop condition will be *true*. If the user types *S*, the first expression (`option != 'S'`) is *false*, but the second expression (`option != 's'`) is *true*. Thus, the loop condition evaluates to true and the *while* loop body is executed. Similarly, if the user types *s*, the first expression (`option != 'S'`) is *true*, so the loop condition evaluates to *true* and the *while* loop body is executed.

An easy method for constructing a correct *while* loop condition consists of three steps:

1. Define the loop termination condition; that is, define the condition that will make the loop stop executing.
2. Create the loop continuation condition—the condition that will keep the loop executing—by applying the logical NOT operator (!) to the loop termination condition.
3. Simplify the loop continuation condition by applying DeMorgan's Laws, where possible.
   DeMorgan's Laws are the following:
   NOT(A AND B) = (NOT A) OR (NOT B)
   and
   NOT(A OR B) = (NOT A) AND (NOT B)

Let's use these three steps to construct the correct loop condition for the menu program.

1. Define the loop termination condition:
   The loop will stop executing when the user enters an *S* or the user enters an *s*. Translating that into Java, we get

   ```
   ( option == 'S' || option == 's' )
   ```

   Description
2. Create the loop continuation condition by applying the ! operator:

```
! ( option == 'S' || option == 'S' )
```

Description

3. Simplify by applying DeMorgan’s Laws:

   To apply DeMorgan’s Laws, we change the == equality operators to != and change the logical OR operator (`||`) to the logical AND operator (`&&`), producing an equivalent, but simpler expression:

```
( option != 'S' && option != 's' )
```

Description

We now have our loop condition.

To illustrate, let’s write an application that calculates the cost of an order at Bonnie’s Burgers. We provide a menu with prices, and the user selects items from the menu one at a time until the user enters *S* or *s* to stop. This is an accumulation operation because we are accumulating the cost of the user’s order. **Example 6.9** shows the code for this application, and **Figure 6.12** shows the output of a sample run.

Example 6.9 A Compound Loop Condition

```java
/* Order from a menu
   Anderson, Franceschi
*/

import java.util.Scanner;
import java.text.DecimalFormat;

public class BonniesBurgers
{
   public static void main( String [ ] args )
   {
      String menu = "\tC Cheeseburger: $7.49";
             menu += "\n\tH Hot dog: $6.99";
             menu += "\n\tL Lemonade: $2.50";
             menu += "\n\tT Iced tea: $2.75";

      DecimalFormat money = new DecimalFormat( "$#.00" );
      Scanner scan = new Scanner( System.in );

      System.out.println( "Welcome to Bonnie's Burgers. "
                           + "Select from our menu: " );

      System.out.println( menu ); // print the menu
      System.out.print( "Order an item, "
                         + "or \"S\" to stop > " );
      char option = scan.next( ).charAt( 0 );
      double orderCost = 0.0;

      while ( option != 'S' && option != 's' )
      {
        switch ( option )
        {
          case 'c', 'C' ->
          {
            System.out.print( "Cheeseburger ordered. " );
            orderCost += 7.49;
          }
          case 'h', 'H' ->
          {
            System.out.print( "Hot dog ordered. " );
            orderCost += 6.99;
          }
          case 'l', 'L' ->
          {
            System.out.print( "Lemonade ordered. " );
            orderCost += 2.50;
          }
          case 't', 'T' ->
          {
            System.out.print( "Iced tea ordered. " );
            orderCost += 2.75;
          }
          default ->
             System.out.println( "Unrecognized menu item." );
        }

        System.out.println( "Subtotal: "
                             + money.format( orderCost ) );

        System.out.print( "\nOrder another item, "
                           + "or \"S\" to stop > " );
        option = scan.next( ).charAt( 0 );
      }

      System.out.println( "\nTotal order cost is "
                            + money.format( orderCost ) );
  }
}
```

Description

Figure 6.12 Ordering from a Menu

Description

We use the compound condition in the *while* loop (line 29). Then within the *while* loop, we use a *switch* statement (lines 31–55) to determine which menu item the user has chosen. We handle case-insensitive input by including *case* constants for both the lowercase and uppercase versions of each letter option. After the *switch* statement, we include the update read (lines 60–62) to input the user’s next item.

Note that we don’t provide *case* statements for the sentinel values. Instead, we use the *while* loop condition to detect when the user enters the sentinel values.

6.7 Testing Techniques for *while* Loops

It's a good feeling when our code compiles without errors. Getting a clean compile, however, is only part of the job for the programmer. The other part of the job is verifying that the code is correct; that is, that the program produces accurate results.

It usually isn't feasible to test a program with all possible input values, but we can get a reasonable level of confidence in the accuracy of the program by concentrating our testing in three areas:

1. Does the program produce correct results with a set of known inputs?
2. Does the program produce correct results if the sentinel value is the first and only input?
3. Does the program deal appropriately with invalid input?

Let's take a look at these three areas in more detail:

1. Does the program produce correct results with known input?

To test the program with known input, we select valid input values and determine what the results should be by performing the program's operation either by hand or by using a calculator. For example, to test whether a total or average is computed correctly, enter some values and compare the program's output to a total or average we calculate by entering those same values into a calculator.

It's especially important to select input values that represent boundary conditions, that is, values that are the lowest or highest expected values. For example, to test a program that determines whether a person is old enough to vote in a presidential election (that is, the person is 18 or older), we should select test values of 17, 18, and 19. These values are the boundary conditions for `age >= 18`; the test values are one integer less, the same value, and one integer greater than the legal voting age. We then run the program with the three input values and verify that the program correctly identifies 17 as an illegal voting age and 18 and 19 as legal voting ages.

1. Does the program produce correct results if the sentinel value is the first and only input?

In our *while* loops, when we find the sentinel value, the flow of control skips the *while* loop body and picks up at the statement following the *while* loop. When the sentinel value is the first input value, our *while* loop body does not execute at all. We simply skip to the statement following the *while* loop. In cases like this, the highly respected computer scientist Donald Knuth recommends that we “do exactly nothing, gracefully.”

In many programs that calculate a total or an average for the input values, when no value is input, our program should either report the total or average as 0 or output a message that no values were entered. It’s important to write our program so that it tolerates the only input being the sentinel value; therefore, we need to test our programs by entering the sentinel value first.

Let’s revisit the earlier examples in this chapter to see how they handle the case when only the sentinel value is entered.

In the addition calculator (Example 6.3), we set the total to 0 before the *while* loop and simply report the value of total after the *while* loop. So we get the correct result (0) with only the sentinel value.

In Example 6.4 where we count the percentage of passing test scores, we handle the sole sentinel value by performing some additional checking after the *while* loop. If only the sentinel value is entered, the count will be 0. We check for this case and if we find a count of 0, we skip reporting the percentage so that we avoid dividing by 0. We use similar code in Example 6.5, where we calculate the average test score. If we detect a count of 0, we also skip the calculation of the average to avoid dividing by 0 and simply report the class average as 0.

1. Does the program deal appropriately with invalid input?

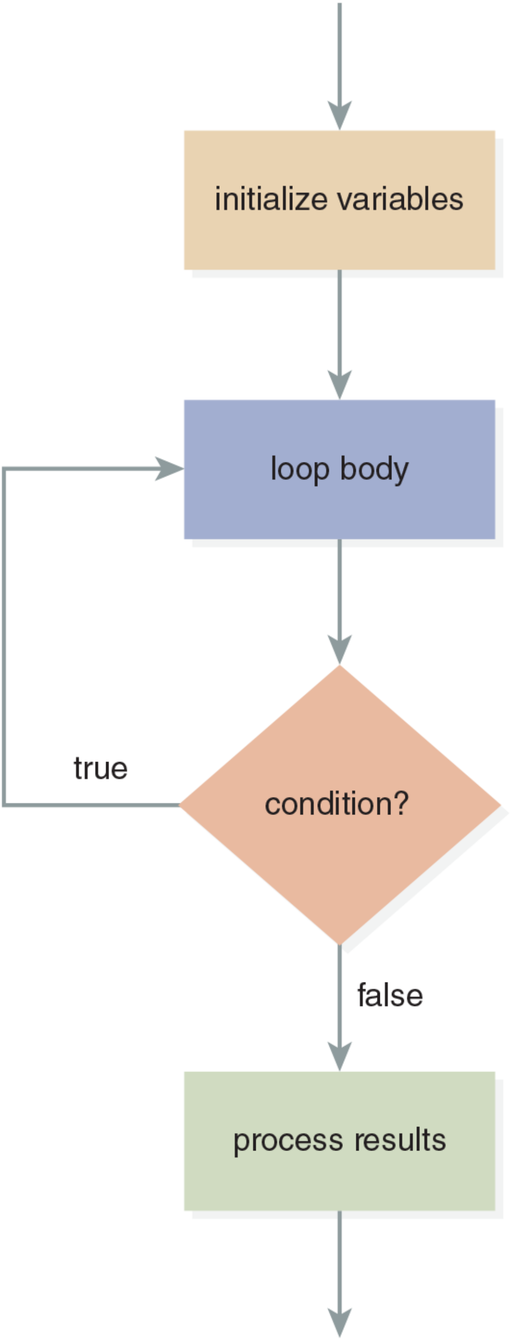
If the program expects a range of values or certain discrete values, then it should notify the user when the input doesn't fit the expected values.

In Example 6.9, we implemented a menu for ordering food items. The user could enter *c, h, l, or t* (or the corresponding capital letters) representing their desired options. If the user entered a letter other than those expected values, we used the *default* clause of the *switch* statement to issue an error message, "*Unrecognized menu item*."

In the next section, we explain how to validate that user input is within a range of values using a *do/while* loop.

SOFTWARE ENGINEERING TIP

Expect that the user might enter the sentinel value first. The program needs to handle this special case.

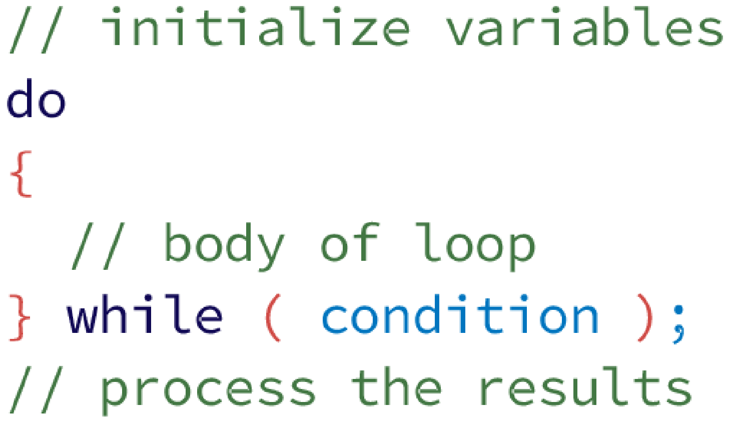

initialize variables
loop body
true
condition?
false
process results

```
Enter a number between 1 and 10 > 20
Enter a number between 1 and 10 > -1
Enter a number between 1 and 10 > 0
Enter a number between 1 and 10 > 11
Enter a number between 1 and 10 > 5
Thank you!
```

6.8 Event-Controlled Loops Using *do/while*

Another form of loop that is especially useful for validating user input is the *do/while* loop. In the *do/while* loop, the loop condition is tested at the end of the loop (instead of at the beginning, as in the *while* loop). Thus the body of the *do/while* loop is executed at least once.

The syntax of the *do/while* loop is the following:



```
// initialize variables
do
{
  // body of loop
} while ( condition );
// process the results
```

Description

Figure 6.13 shows the flow of control of a *do/while* loop.

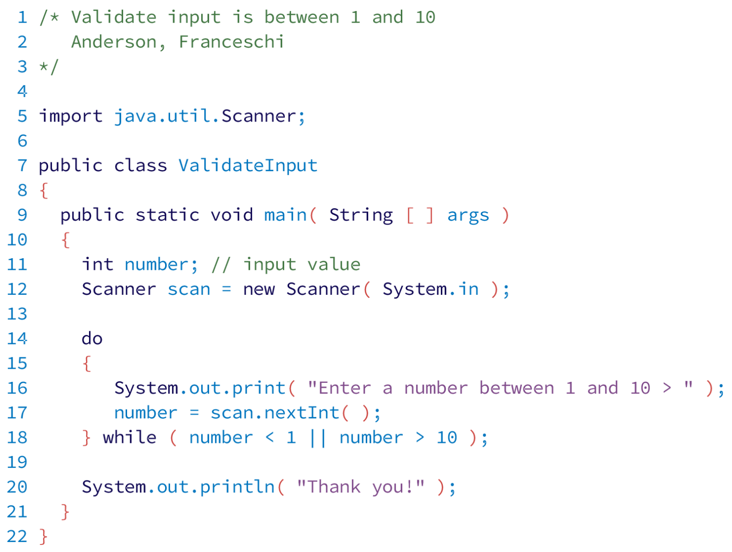
Figure 6.13 The Flow of Control of a *do/while* Statement

Description

To use the *do/while* loop to validate user input, we insert the prompt for the input inside the body of the loop, then use the loop condition to test the value of the input. Like the *while* loop, the body of the loop will be reexecuted if the condition is *true*. Thus, we need to form the condition so that it’s *true* when the user enters invalid values. Be aware that because of scope, any variable that will be referenced in the *while* condition must be defined before the *do/while* loop begins.

Example 6.10 implements a *do/while* loop (lines 14–18) that prompts the user for an integer between 1 and 10. **Figure 6.14** shows the output of the program. If the user enters a number outside the valid range, we reprompt the user until the input is between 1 and 10. Thus, the condition for the *do/while* loop (line 18) checks whether the number is less than 1 or greater than 10. Note also that because we will be referencing *number* in the *while* condition, we must define *number* before the *do/while* loop begins (line 11).

Example 6.10 Validating User Input



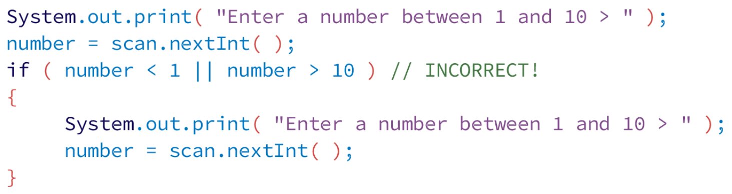
```java
/* Validate input is between 1 and 10
   Anderson, Franceschi
*/

import java.util.Scanner;

public class ValidateInput
{
  public static void main( String [ ] args )
  {
    int number; // input value
    Scanner scan = new Scanner( System.in );

    do
    {
       System.out.print( "Enter a number between 1 and 10 > " );
       number = scan.nextInt( );
    } while ( number < 1 || number > 10 );

    System.out.println( "Thank you!" );
  }
}
```

Description

Figure 6.14 Validating Input

Description

For validating input, we may be tempted to use an *if* statement rather than a *do/while* loop. For example, to perform the same validation as Example 6.10 we may try this **incorrect** code:



```java
System.out.print( "Enter a number between 1 and 10 > " );
number = scan.nextInt( );
if ( number < 1 || number > 10 ) // INCORRECT!
{
     System.out.print( "Enter a number between 1 and 10 > " );
     number = scan.nextInt( );
}
```

Description

The problem with this approach is that the *if* statement will reprompt the user only once. If the user enters an invalid value a second time, the program will not catch it. A *do/while* loop, however, will continue to reprompt the user as many times as needed until the user enters a valid value.

COMMON ERROR TRAP

Do not use an *if* statement to validate input because it will catch invalid values entered the first time only. Use a *do/while* loop to reprompt the user until the user enters a valid value.

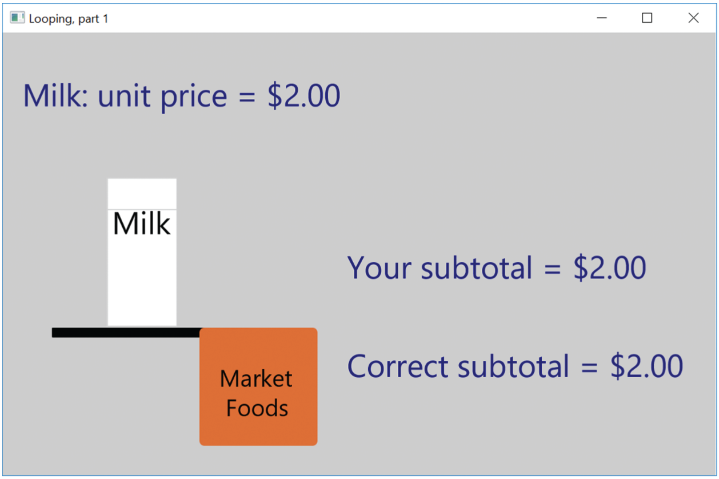
with these end-of-chapter questions

6.14.2	Reading and Understanding Code Questions 9, 10
6.14.3	Fill In the Code Question 28
6.14.4	Identifying Errors in Code Question 34
6.14.5	Debugging Area Question 36
6.14.6	Write a Short Program Question 53

CODE IN ACTION

To see two step-by-step illustrations of *do/while* loops, look for the movie within the online resources. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock



Milk: unit price = $2.00

Your subtotal = $2.00

Correct subtotal = $2.00

6.9 Programming Activity 1: Using *while* Loops

In this activity, you will work with a sentinel-controlled *while* loop, performing this activity:

> Write a *while* loop to process the contents of a grocery cart and calculate the total price of the items. It is important to understand that, in this example, we do not know how many items are in the cart.

The framework will animate your code and display the current subtotal so that you can check the correctness of your code. The window will display the various *Item* objects moving down a conveyor belt toward a grocery bag. It will also display the unit price of the item and your current subtotal, as well as the correct subtotal.

For example, **Figure 6.15** demonstrates the animation: We are currently scanning the first item, a milk carton, with a unit price of $2.00; thus, the correct subtotal is $2.00.

Figure 6.15 Animation of the *Cashier* Application

Description

As the animation will show, *Item* objects could be milk, cereal, orange juice, or the divider bar. The number of *Item* objects in the cart is determined randomly; as you watch the animation, sometimes you will find that there are two items in the cart, sometimes six, sometimes three, sometimes only the divider bar, and so forth. Scanning the divider bar signals the end of the items in the cart.

Task Instructions

Copy the files in this chapter’s Programming Activity 1 folder in the source code provided with this text to a folder on your computer. Searching for five stars (*****) in the *CashierDrawing.java* source code will show you where to add your code. You will add your code inside the *checkout* method of the *CashierDrawing* class (the method header for the *checkout* method has already been coded for you). **Example 6.11** shows a fragment of the *CashierDrawing* class, where you will add your code:

Example 6.11 The *checkout* Method in *CashierDrawing.java*

```
public void checkout( )
{
 /* ***** Student writes the body of this method ***** */
 //
 //  Using a while loop, calculate the total price
 //  of the groceries.
 //
 // The getNext method (in this CashierDrawing class) returns the
 // next item on the conveyor belt, which is an Item object
 // (we do not know which item and we do not know how many items
 // are in the cart - this is randomly generated).
 // getNext does not take any arguments. Its API is:
 //       Item getNext( )
 //
 // Right after you update the current subtotal,
 // you should call the animate method.
 // The animate method takes one argument: a double,
 // which is your current subtotal.
 // For example, if the name of your variable representing
 // the current subtotal is total, your call to the animate
 // method should be:
 //    animate( total );
 //
 // The instance method getPrice of the Item class
 // returns the price of the Item object.
 // The method getPrice does not take any arguments.
 // Its API is:
 //        double getPrice( )
 //
 // The cart is empty when the getNext method returns
 // the divider Item.
 // You detect the divider Item because its price
 // is -0.99. So an Item with a price of -0.99
 // is the sentinel value for the loop.
 //
 // After you scan the divider, print the total to the console.

 // End of student code
}
```

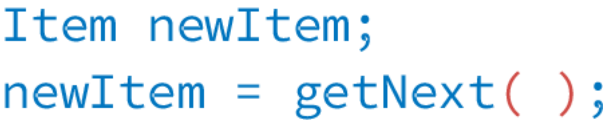
Description

- You can access items in the cart by calling the *getNext* method of the *CashierDrawing* class, which has the following API:

```
Item getNext( )
```

Description

The *getNext* method returns an *Item* object, which represents an *Item* in the cart. As you can see, the *getNext* method does not take any arguments. Since we call the method *getNext* from inside the *CashierDrawing* class, we call the method without an object reference. For example, a call to *getNext* could look like the following:

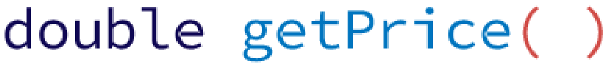


```
Item newItem;
newItem = getNext( );
```

Description

The *getNext* method is already written and contains code to randomly generate *Items*. It is written in such a way that the first *Item* object on the conveyor belt may or may not be the divider. (If the first *Item* is the divider, the cart is empty.)

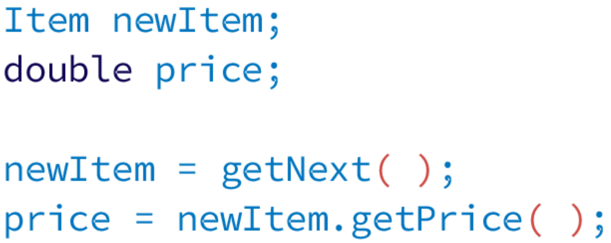
- After you get a new *Item*, you can “scan” the item to get its price by calling the *getPrice* method of the *Item* class. The *getPrice* method has this API:



```
double getPrice( )
```

Description

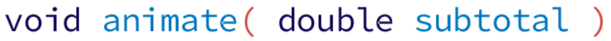
Thus, you would get an item, then get its price using code like the following:



```
Item newItem;
double price;

newItem = getNext( );
price = newItem.getPrice( );
```

Description

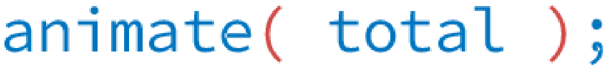
- After adding the price of an item to your subtotal, call the *animate* method of the *CashierDrawing* class. This method will display both your subtotal and the correct subtotal so that you can verify that your code is correct.
  The animate method has the following API:



```
void animate( double subtotal )
```

Description

Thus, if your variable representing the current total is *total*, you would call the animate method using the following code:



```
animate( total );
```

- You want to exit the loop when the next *Item* is the divider. You will know that the *Item* is the divider because its price will be –0.99 (negative 0.99); thus, scanning an *Item* whose price is –0.99 should be your condition to exit the *while* loop.
- When your loop ends, verify that your total matches the correct subtotal displayed.
- To test your code, compile *CashierDrawing.java* and run the application from the *CashierApplication* class.

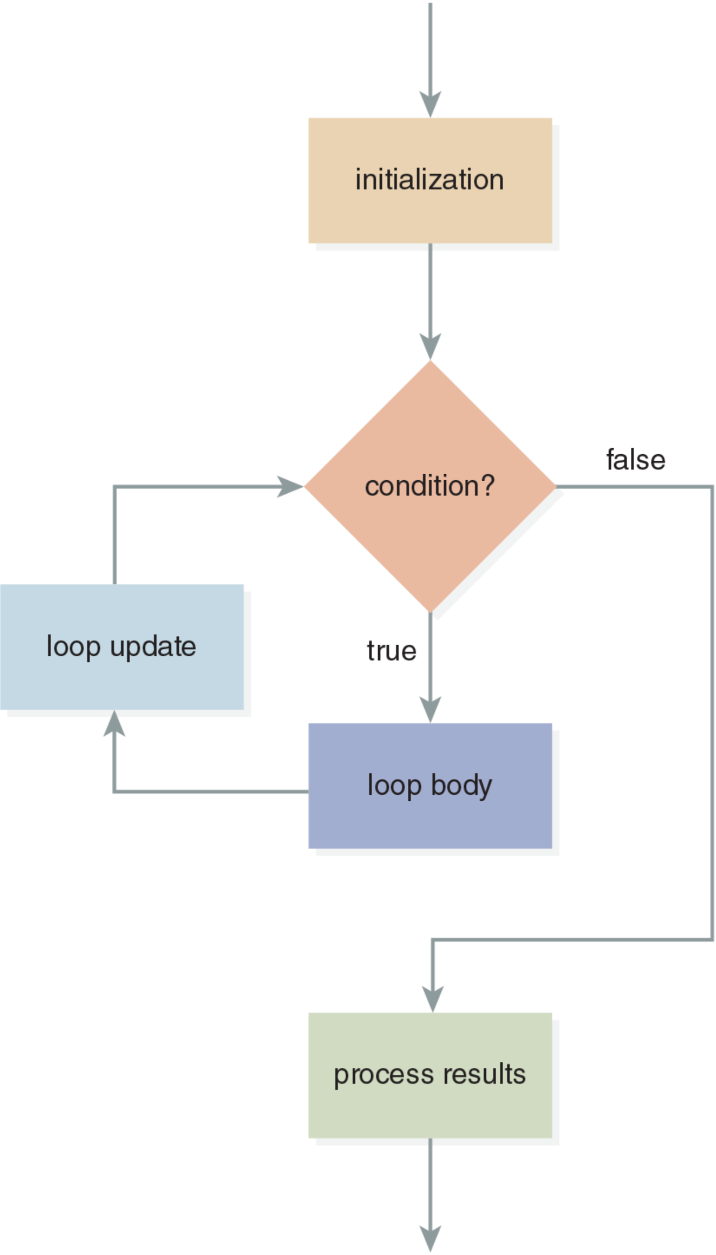
Troubleshooting

If your method implementation does not animate or animates incorrectly, check these items:

- Verify that you have correctly coded the priming read.
- Verify that you have correctly coded the condition for exiting the loop.
- Verify that you have correctly coded the body of the loop.

DISCUSSION QUESTIONS

1. What is the sentinel value of your *while* loop?
2. Explain the purpose of the priming read.


initialization
condition?
false
true
loop update
loop body
process results

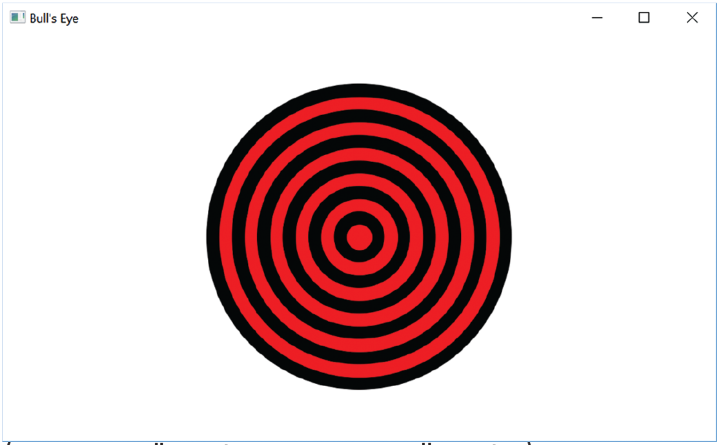
```
Enter an integer > 12
Enter an integer > 10
Enter an integer > 5
Enter an integer > 7
Enter an integer > 3
The total is 37
```

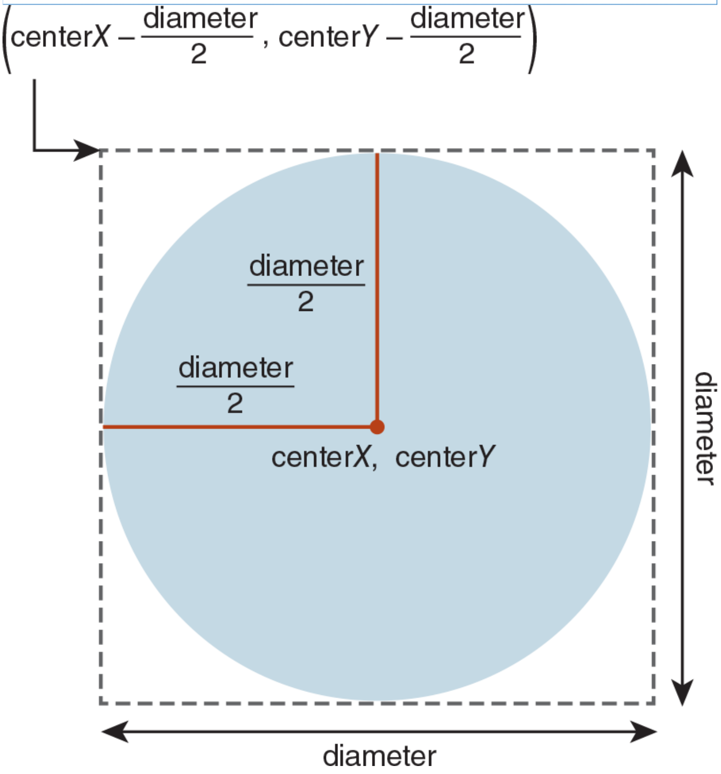
```
0 2 4 6 8 10 12 14 16 18 20
```

```
Enter integer 1 > 23
Enter integer 2 > 12
Enter integer 3 > 10
Enter integer 4 > 11
Enter integer 5 > 15
The total is 71
```

```
Enter a sentence:
Programming is not a spectator   sport!
The sentence contains 6 tokens.
```

```
Enter a sentence:
Programming is not a spectator sport!
The sentence backwards is:
!trops rotatceps a ton si gnimmargorP
```


Bull's Eye


$\left(\text{center}X - \frac{\text{diameter}}{2}, \text{center}Y - \frac{\text{diameter}}{2}\right)$
$\frac{\text{diameter}}{2}$
$\frac{\text{diameter}}{2}$
center$X$, center$Y$
diameter
diameter

```
Enter integer 1 > 23
Enter integer 2 > 12
Enter integer 3 > 10
Enter integer 4 > 11
Enter integer 5 > 15
The total is 71
```

```
Enter a sentence:
The sentence backwards is
```

6.10 Count-Controlled Loops Using *for*

Before the loop begins, if we know the number of times the loop body should execute, we can use a **count-controlled loop**. The *for* loop is designed for count-controlled loops, that is, when the number of iterations is determined before the loop begins.

6.10.1 Basic Structure of *for* Loops

The *for* loop has this syntax:

```
for ( initialization; loop condition; loop update )
{
        // loop body
}
```

Description

COMMON ERROR TRAP

Be sure to use semicolons, rather than commas, to separate the statements in a *for* loop header.

Notice that the initialization, loop condition, and loop update in the *for* loop header are separated by semicolons (not commas). Notice also that there is no semicolon after the closing parenthesis in the *for* loop header. A semicolon here would indicate an empty *for* loop body. Although some advanced programs might correctly write a *for* loop with an empty loop body, the programs we write in this text will have at least one statement in the *for* loop body.

The scope of any variable declared within the *for* loop header or body extends from the point of declaration to the end of the *for* loop body.

COMMON ERROR TRAP

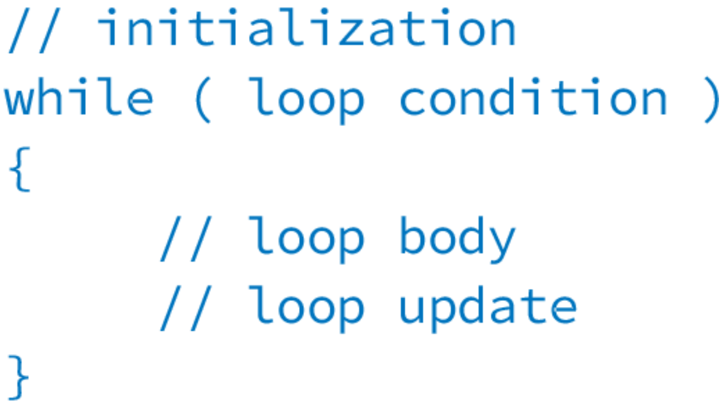
Adding a semicolon after the closing parenthesis in the *for* loop header indicates an empty loop body and will likely cause a logic error.

The flow of control of the *for* loop is shown in **Figure 6.16**. When the *for* loop is encountered, the initialization statement is executed. Then the loop condition is evaluated. If the condition is true, the loop body is executed, then the loop update statement is executed, and the loop condition is reevaluated. Again, if the condition is true, the loop body is executed, followed by the loop update, then the reevaluation of the condition, and so on, until the condition is false.

Figure 6.16 Flow of Control of the *for* Loop

Description

The *for* loop is equivalent to the following *while* loop:



```
// initialization
while ( loop condition )
{
      // loop body
      // loop update
}
```

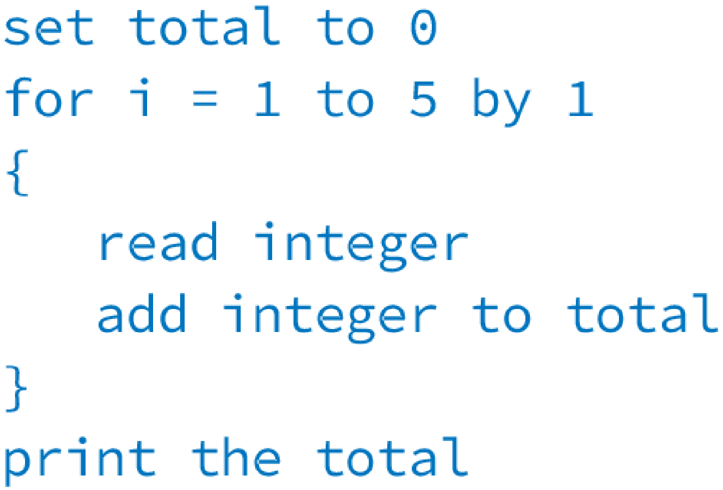
Description

As we can see, *while* loops can be used for either event-driven or count-controlled loops. A *for* loop is especially useful for count-controlled loops, however. Because all the loop control is contained in the *for* loop header, we can easily see what condition will stop the loop and how the condition will be updated after each iteration.

6.10.2 Constructing *for* Loops

Typically, we use a **loop control variable** in a *for* loop; that control variable is usually used for counting. We set its initial value in the initialization statement, increment or decrement its value in the loop update statement, and check its value in the loop condition.

For example, if we want to find the sum of five integers, we know the loop body should execute five times—once for each integer. We set our loop control variable to 1 in the initialization statement, increment the loop control variable by 1 in the loop update statement, and check if its value is less than or equal to 5 in the loop condition. The pseudocode for this program is the following:



```
set total to 0
for i = 1 to 5 by 1
{
   read integer
   add integer to total
}
print the total
```

Description

With a *for* loop, we do not need to perform a priming read because the condition for exiting the loop is controlled by a counter, not by an input value.

Example 6.12 shows the *for* loop for calculating the sum of five integers.

Example 6.12 Finding the Sum of Five Numbers

```
 1 /* Find the total of 5 numbers
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6
 7 public class Sum5Numbers
 8 {
 9   public static void main( String [ ] args )
10   {
11     int total = 0;    // stores the sum of the 5 numbers
12     int number;       // stores the current input
13
14     Scanner scan = new Scanner( System.in );
15
16     for ( int i = 1; i <= 5; i++ )
17     {
18       System.out.print( "Enter an integer > " );
19       number = scan.nextInt( );
20
21       total += number;  // add input to total
22     }
23
24     // process results by printing the total
25     System.out.println( "The total is " + total );
26   }
27 }
```

Description

In this example, which is a standard accumulation operation, the *for* loop initialization statement declares *i*, which will be our loop control variable. We start *i* at 1, and after each execution of the loop body, we increment *i* by 1 in the loop update statement. The loop condition checks if the value of *i* is less than or equal to 5; when *i* reaches 6, we have executed the loop body five times. **Figure 6.17** shows the execution of this *for* loop.

Figure 6.17 Finding the Sum of Five Integers

Description

Note that because we declare our loop counter variable *i* in the *for* loop header, we cannot reference *i* after the *for* loop ends. Thus, this code would generate a compiler error, because *i* is out of scope on line 24:

```
16    for ( int i = 1; i <= 5; i++ )
17    {
18      System.out.print( "Enter an integer > " );
19      number = scan.nextInt( );
20
21      total += number;  // add input to total
22    }
23
24    System.out.println( "The total for the " + ( i - 1 )
25                        + " numbers is " + total ); // i is out of scope
```

Description

Defining a new variable using the same name as a variable already in scope is invalid and generates a compiler error. However, a variable name can be reused when a previously defined variable with the same name is no longer in scope. In the code above, the scope of the variable *i* defined in line 16 is limited to the *for* loop on lines 16–22. We cannot define another variable named *i* in that *for* loop; however, as shown below, we could reuse the name *i* in a subsequent *for* loop (lines 24–29), because the first *i* is no longer in scope. In fact, programmers often reuse the variable name *i* for the counter variable in their *for* loops.

```
16    for ( int i = 1; i <= 5; i++ )
17    {
18      System.out.print( "Enter an integer > " );
19      number = scan.nextInt( );
20
21      total += number; // add input to total
22    }
23
24    for ( int i = 1; i <= 10; i++ )
25    {
26      System.out.print( "Enter integer " + i + " > " );
27      number = scan.nextInt( );
28    }
```

Description

If we do want to refer to the loop variable after the loop ends, we can define the variable before the *for* loop, as shown in the following code:

```
    int i;
    for ( i = 1; i <= 5; i++ )
    {
      System.out.print( "Enter an integer > " );
      number = scan.nextInt( );

      total += number; // add input to total
    }

    System.out.println( "The total for " + ( i - 1 ) // i is 6
                         + " numbers is " + total );
```

Description

We can also increment the loop control variable by values other than 1. **Example 6.13** shows a *for* loop that increments the control variable by 2 to print the even numbers from 0 to 20.

The pseudocode for this program is the following:

```
set output to an empty String
for i = 0 to 20 by 2
{
    append i and a space to the output String
}
print the output String
```

Description

Example 6.13 Printing Even Numbers

```
/* Print the even numbers from zero to twenty
   Anderson, Franceschi
*/

public class PrintEven
{
  public static void main( String [ ] args )
  {
    String toPrint = ""; // initialize output String

    for ( int i = 0; i <= 20; i += 2 )
    {
       toPrint += i + " "; // append current number and a space
    }

    System.out.println( toPrint ); // print results
  }
}
```

Description

We start with an empty *String* variable, *toPrint*, and with each iteration of the loop we append the next even number and a space. When the loop completes, we output *toPrint*, which prints all numbers on one line, as shown in **Figure 6.18**.

Figure 6.18 Printing Even Numbers from 0 to 20

Description

In this example, we initialize the loop control variable to 0, then increment *i* by 2 in the loop update statement (`i += 2`) to skip the odd numbers. Notice that we used the value of the loop control variable *i* inside the loop, illustrating that the loop control variable can perform double duty such as being the counter and the data.

The loop control variable also can be used in our prompt to the user. For example, in Example 6.12 we could have prompted the user for each integer using this statement:

```
System.out.print( "Enter integer " + i + " > " );
```

Then the user’s prompt would look like that shown in **Figure 6.19**.

Figure 6.19 Adding the Loop Control Variable to the Prompt

Description

CODE IN ACTION

To see a step-by-step illustration of a *for* loop, look for the movie within the online resources. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

A *for* loop can be especially useful for processing a *String* character by character. We define the loop control variable so that it corresponds to the index of each character in the *String.* We initialize the loop control variable to 0 (the index of the first character in a *String*), increment the loop control variable by 1 in the loop update, and set the condition of the *for* loop to be that the loop control variable is less than the length of the *String*. In each iteration of the loop, we process one character of the *String* by extracting the character at the loop control variable using the *charAt* method of the *String* class.

Example 6.14 demonstrates this technique. In this example, we prompt the user for a sentence and count the number of tokens in the sentence. A token is defined as a sequence of non-white space characters delimited by one or more white space characters; that is, spaces or tabs.

Example 6.14 Counting Tokens

```java
/* Count tokens in a sentence
*  Anderson, Franceschi
*/

import java.util.Scanner;
public class CountTokens
{
  public static void main( String [ ] args )
  {
     Scanner scan = new Scanner( System.in );

     System.out.println( "Enter a sentence:" );
     String sentence = scan.nextLine( );
     sentence = sentence.trim( ); // remove leading/trailing white space

     if ( sentence.length( ) == 0 )
     {
        System.out.println( "The sentence is empty." );
     }
     else
     {
         boolean inWhiteSpace = false;
         int countTokens = 1;

         for ( int i = 0; i < sentence.length( ); i++ )
         {
            char c = sentence.charAt( i );
            if ( c == ' ' || c == '\t' )
            {
              if ( ! inWhiteSpace )
              {
                  countTokens++;
                  inWhiteSpace = true;
              }
            }
            else // not a white space character
            {
              inWhiteSpace = false;
            }

         }
         System.out.println( "The sentence contains " + countTokens
                      + ( countTokens == 1 ? " token." : " tokens." ) );
     }
  }
}
```

Description

To count the tokens, we look at each character in the sentence, one at a time. If a character is not a white space character, then it is part of a token. When we find a white space character, however, it is a signal that the token is complete. At that time, we increment our token count.

Tokens may be separated by more than one white space character, however, so we need to detect when we find multiple consecutive white space characters and ignore all but the first white space character. To do this, we define a *boolean* variable, called a **flag**. We set the flag to true when we see a white space character and to false when we see a non-white space character. If we find a white space character when the flag is false, it must be the first white space character. If the flag is set to true, however, we have found an extra white space character, which we should ignore.

The pseudocode for this example is

```
read a sentence
remove leading and trailing white space
if sentence is empty
       print message no tokens found
else
     set countTokens = 1
    initialize flag variable to false to indicate that the current character is not
    white space
    for i = 0 to (sentence length -1) by 1
   {
        get character at position i in sentence
        if ( character is a space or tab )
             if flag variable is false
                   we have found the end of a token, so increment countTokens
                  set flag variable to true to indicate that current character is
                  white space
        else
            set flag variable to false to indicate that current character is not
            white space
   }
   print the number of tokens found
```

Description

Example 6.14 shows our code. We prompt the user for a sentence using *Scanner*'s *nextLine* method, which reads all characters on the line including leading and trailing white space. We want to remove those extraneous white space characters, so we call the *trim* method of the *String* class (line 14), shown in **Table 6.4**, which will remove any leading and trailing white space for us.

The *trim* Method of the *String* Class

The *trim* Method of the *String* Class

Return value	Method name and argument list
`String`	`trim( )` returns a *String* with leading and trailing white space characters removed

Description

It is possible that the user pressed the *Enter* key without entering any characters (or only white space characters). In that case, the trimmed sentence will be empty. We check for an empty *String* on line 16, and if true, we output a message and do nothing more.

If, however, the user has entered some characters, we initialize our flag variable, *inWhiteSpace,* to *false* (line 22), because we have indeed not processed any characters and thus have not found white space.

We detect tokens by finding the white space following the token, but when we finish processing the sentence, the last character will not be white space because we trimmed the sentence. Thus we will have one more token than the number of times we found white space. To account for that last token, we initialize our *tokenCount* variable to 1 (line 23).

Now we are ready for our *for* loop (lines 25–41), which processes each character in the sentence looking for white space and counting tokens. As mentioned, the loop control variable*, i*, corresponds to the index of the current character being processed. We initialize *i* to 0, the first index of the *String*; increment *i* by 1 in the loop update; and check that *i* is less than *sentence.length( )* in the loop condition (line 25). We extract the character at *i* using the *charAt* method (line 27).

In line 28, we check whether the current character is a space or a tab. If so, we have detected white space. If the flag is not set to *true* (line 30), then this is the first white space character in a possible sequence of white space characters, and we count the token, then set the flag to *true* so that we ignore any succeeding white space characters (lines 32–33). If the current character is not white space, we set the flag to *false* to indicate that we are inside a token (line 38).

When the loop ends, we print a message with the token count (lines 42–43). **Figure 6.20** shows the output when the user enters a sentence with six tokens separated by spaces and a tab character.

Figure 6.20 Output from Example 6.14

Description

We can also decrement the loop control variable. **Example 6.15** shows an application that reads a sentence entered by the user and prints the sentence backward.

Example 6.15 Printing a Sentence Backwards

```java
/* Print a sentence backwards
   Anderson, Franceschi
*/
import java.util.Scanner;

public class Backwards
{
  public static void main( String [ ] args )
  {
    Scanner scan = new Scanner( System.in );

    String backwards = "";

    System.out.println( "Enter a sentence:" );
    String original = scan.nextLine( );

    for ( int i = original.length( ) - 1; i >= 0; i-- )
    {
        backwards += original.charAt( i );
    }

    System.out.println( "The sentence backwards is: \n" + backwards );
  }
}
```

Description

The pseudocode for this program is the following:

```
set backwards to an empty String
read a sentence

for i = ( length of sentence - 1 ) to 0 by -1
{
   get character at position i in sentence
   append character to backwards
}
print backwards
```

Description

To print a sentence backward, we treat the sentence, a *String*, like a stream of characters; each iteration of the loop extracts and processes one character from the *String*, using the *charAt* method of the *String* class. Line 12 declares a *String*, *backwards* (initialized as an empty *String*), to hold the reverse of the user's sentence. Lines 14–15 prompt the user for a sentence. Lines 17 through 20 make up the *for* loop, whose purpose is to copy the original sentence into the *String* *backwards* with its characters in reverse order. We do this by starting the copying at the last character in the *original String* and moving backward one character at a time until we have copied the first character. Thus, we initialize our loop variable to the position of the last character in *original* (`original.length( ) – 1`) and extract one character at a time, appending it to *backwards*. The loop update statement (`i––`) moves the loop variable backward by one position, and our loop condition (`i >= 0`) checks whether we have reached the beginning of the *String original*. **Figure 6.21** shows the execution of the program with the user entering the sentence, "*Programming is not a spectator sport!*"

Figure 6.21 Printing a Sentence Backward

Description

We can display some interesting graphics using *for* loops. The graphical application in **Example 6.16** draws the bull's-eye target shown in **Figure 6.22**. To make the bull's-eye target, we draw 12 concentric circles (circles that have the same center point), beginning with the largest circle and successively drawing a smaller circle on top of the circles already drawn. Thus, the bull's-eye target circles have the same center point, but different diameters. The pseudocode for this program is

```
for diameter = 300 to 25 by -25
{
    draw the circle with current diameter
    if color is black
         set color to red
    else
         set color to black
}
```

Description

Translating the pseudocode into Java, we get the code shown in Example 6.16.

Example 6.16 Drawing a Bull's Eye

```
 1 /* Bull's-eye Target
 2    Anderson, Franceschi
 3 */
 4
 5 import javafx.application.Application;
 6 import javafx.scene.canvas.Canvas;
 7 import javafx.scene.canvas.GraphicsContext;
 8 import javafx.scene.paint.Color;
 9 import javafx.stage.Stage;
10
11 public class BullsEye extends Application
12 {
13
14     @Override
15     public void start( Stage stage )
16     {
17         // set up window title and size
18         GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
19                   stage, "Bull's Eye", 700, 400 );
20
21        // center of the bull's eye
22        final int CENTER_X = 350, CENTER_Y = 200;
23
24        // start and end diameter, amount to decrease diameter
25        final int START_DIAMETER = 300, END_DIAMETER = 25,
26                  DECREMENT = 25;
27
28        // color of first circle
29        Color toggleColor = Color.BLACK;
30        gc.setFill( toggleColor );
31
32        for ( int diameter = START_DIAMETER; diameter >= END_DIAMETER;
33                  diameter -= DECREMENT )
34        {
35           // draw circle with current diameter value
36           gc.fillOval( CENTER_X - diameter / 2, CENTER_Y - diameter / 2,
37                        diameter, diameter );
38           // switch color
39           if ( toggleColor.equals( Color.BLACK ) )
40                toggleColor = Color.RED;
41           else
42                toggleColor = Color.BLACK;
43
44           gc.setFill( toggleColor );
45
46        }
47     }
48
49     public static void main( String [ ] args )
50     {
51         launch( args );
52     }
53 }
```

Description

Figure 6.22 Drawing a Bull's-Eye

Description

Our *for* loop initialization statement in lines 32–33 sets up the diameter of the largest circle as 300 pixels and the loop update statement decreases the diameter of each circle by 25 pixels. The smallest circle we want to draw should have a diameter of 25 pixels, so we set the loop condition to check that the diameter is greater than or equal to 25. We need to start with the largest circle rather than the smallest circle so that new circles we draw don't hide the previously drawn circles.

Drawing the bull's-eye target circles illustrates two common programming techniques: conversion between units and a toggle variable.

We need to convert between units because the *fillOval* method of the *GraphicsContext* class takes as its arguments the upper-left (*x, y*) coordinate and the width and height of the circle's bounding rectangle. However, all our circles have the same center point, but not the same upper-left *x* and *y* coordinates. Given the diameter and the center point of the circle, however, we can calculate the (*x, y*) coordinate of the upper-left corner. **Figure 6.23** shows how we make the conversion.

Figure 6.23 Drawing a Bull's-Eye

Description

The difference between the center point and the upper-left corner of the bounding rectangle is the radius of the circle, which is half of the diameter (`diameter / 2`). So, the upper-left *x* value is the *x* value of the center point minus half the diameter (`centerX – diameter / 2`). Similarly, the upper-left *y* value is the *y* value of the center point minus half the diameter (`centerY – diameter / 2`).

Thus, we draw each circle using the following statement:

```
gc.fillOval ( centerX - diameter / 2,
              centerY - diameter / 2,
              diameter, diameter );
```

Description

To alternate between red and black circles, we use a **toggle**

variable, which is a variable whose value alternates between two values. We use a *Color* object for our toggle variable, *toggleColor*, and initialize it to *Color.BLACK*. After drawing each circle, we switch the color (lines 38–42). If the current color is black, we set it to red; otherwise, the color must be red, so we set the color to black.

SOFTWARE ENGINEERING TIP

Use a toggle variable when you need to alternate between two values.

with these end-of-chapter questions

6.14.1	Multiple Choice Exercises Questions 2, 3, 4
6.14.2	Reading and Understanding Code Questions 11, 12, 13, 14, 15, 16, 17, 18, 19
6.14.3	Fill In the Code Questions 27, 29
6.14.4	Identifying Errors in Code Questions 32, 33, 35
6.14.5	Debugging Area Questions 38, 39, 40, 41, 42
6.14.6	Write a Short Program Questions 43, 46, 47, 48, 49, 50, 51, 52

6.10.3 Testing Techniques for *for* Loops

One of the most important tests for *for* loops is that the starting and ending values of the loop variable are set correctly. For example, to execute a *for* loop five times, we could set the initial value of the loop variable to 0 and use the condition (`i < 5`), or we could set the initial value of the loop variable to 1 and use the condition (`i <= 5`). Either of these *for* loop headers will cause the loop to execute five times.

```
for ( int i = 0; i < 5; i++ ) // executes 5 times
```

Description

or

```
for ( int i = 1; i <= 5; i++ ) // executes 5 times
```

Description

However, the following *for* loop header is incorrect; the loop will execute only four times.

```
for ( int i = 1; i < 5; i++ ) // INCORRECT! executes only 4 times
```

Description

Thus, to test the *for* loop in Example 6.12 that prompts for five integers, we need to verify that the program outputs exactly five prompts. To test that we are prompting the user five times, we can enter the integers 1, 2, 3, 4, and 5 at the prompts. Another option, shown in **Figure 6.24**, is to append a number to the prompt, which does double duty. Besides keeping the user informed of the number of integers entered so far, it also helps to verify that we have the correct number of prompts.

Figure 6.24 Counting Five Prompts

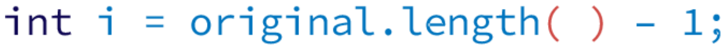
Description

Like *while* loops, the body of a *for* loop may not be executed at all. If the loop condition is *false* the first time it is tested, the body of the *for* loop is skipped. Thus, when testing, we want to simulate input that would cause the loop condition to be *false* when the *for* loop statement is first encountered. For example, in the *Backwards* class in Example 6.15, we need to test the *for* loop with an empty sentence. In other words, when the prompt appears to enter a sentence, we simply press the *Enter* key. If we try this, we will find that the application still works, as **Figure 6.25** shows.

Figure 6.25 The *Backwards* Class with an Empty Sentence

Description

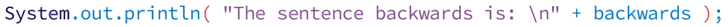
The program works correctly with an empty sentence because the *for* loop initialization statement is



```
int i = original.length( ) - 1;
```

Description

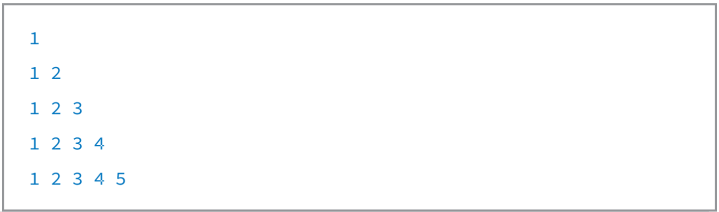
Because the length of an empty *String* is 0, this statement sets *i* to –1. The loop condition (`i >= 0`) is immediately *false*, so the loop body is never executed. The flow of control skips to the statement following the loop,



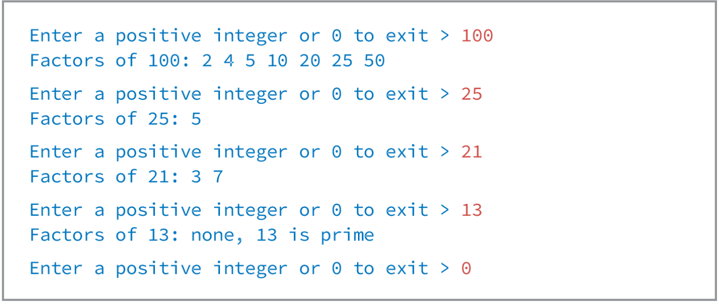
```
System.out.println( "The sentence backwards is: \n" + backwards );
```

Description

which prints an empty *String*. Although it would be more user friendly to check whether the sentence is empty and print a message to that effect, the program does indeed “do exactly nothing, gracefully.”



```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```



```
Enter a positive integer or 0 to exit > 100
Factors of 100: 2 4 5 10 20 25 50
Enter a positive integer or 0 to exit > 25
Factors of 25: 5
Enter a positive integer or 0 to exit > 21
Factors of 21: 3 7
Enter a positive integer or 0 to exit > 13
Factors of 13: none, 13 is prime
Enter a positive integer or 0 to exit > 0
```

6.11 Nested Loops

Loops can be nested inside other loops; that is, the body of one loop can contain another loop. For example, a *while* loop can be nested inside another *while loop* or a *for* loop can be nested inside another *for* loop. In fact, the nested loops do not need to be the same loop type; that is, a *for* loop can be nested inside a *while* loop, and a *while* loop can be nested inside a *for* loop.

Nested loops may be useful if we are performing multiple operations, each of which has its own count or sentinel value. Going back to Jane, our grocery cashier, her workday can be modeled using nested loops. In Programming Activity 1, we wrote the code for our cashier to calculate the total cost of the contents of one customer's grocery cart. But cashiers check out multiple customers, one after another. While the line of people in front of the cashier is not empty, she will help the next customer. For each customer, she will set the total order to $0.00 and start scanning items and add the prices to the total. While the current customer still has items in the cart, Jane will scan the next item. When Jane finishes processing a customer's cart, she will check to see if there is a customer waiting in line. If there is one, she will set the total to $0.00 and start scanning the next customer's items.

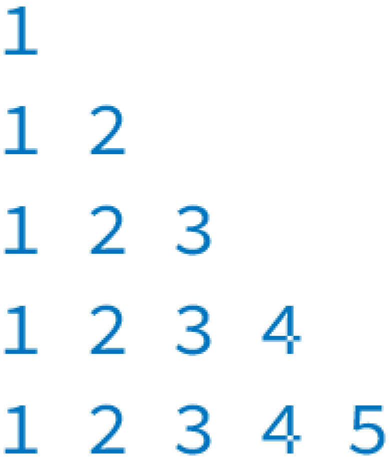
Thus, the cashier's job can be described using a *while* loop nested inside another *while* loop. The pseudocode for these nested loops is shown here:

```
look for a customer
while there is a customer in line
{
  set total to $0.00
  reach for first item
while item is not the divider bar
{
      add price to total
      reach for next item
}
// if we get here, the item is the divider bar
output the total price

look for another customer
```

Description

The important point to understand with nested loops is that the inner (or nested) loop executes completely (executes all its iterations) for each single iteration of the outer loop.

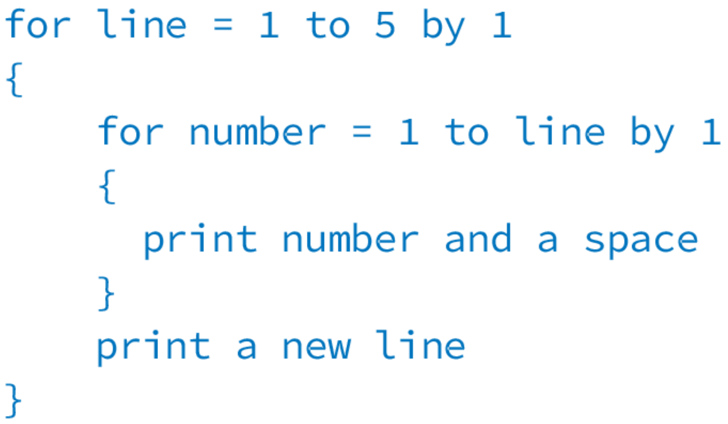
Let's look at a simple example that uses nested *for* loops. Suppose we want to print five rows of numbers as shown here:



```
1

1 2

1 2 3

1 2 3 4

1 2 3 4 5
```

Description

We can see a pattern here. In the first line, we print one number; in the second line, we print two numbers, and so on. In other words, the quantity of numbers we print and the line number are the same. The pseudocode for this pattern is the following:



```
for line = 1 to 5 by 1
{
    for number = 1 to line by 1
    {
      print number and a space
    }
    print a new line
}
```

Description

Translating this pseudocode into nested *for* loops, we get the code shown in **Example 6.17**.

Example 6.17 Nested *for* Loops

```
 1 /*  Printing numbers using nested for loops
 2     Anderson, Franceschi
 3 */
 4
 5 public class NestedForLoops
 6 {
 7   public static void main( String [ ] args )
 8   {
 9      // outer for loop prints 5 lines
10      for ( int line = 1; line <= 5; line++ )
11      {
12        // inner for loop prints one line
13        for ( int number = 1; number <= line; number++ )
14        {
15          // print the number and a space
16          System.out.print( number + " " );
17        }
18
19        System.out.println( ); // print a newline
20      }
21   }
22 }
```

Description

Notice that the inner *for* loop (lines 12–17) uses the value of *line*, which is set by the outer *for* loop (lines 9–20). Thus, for the first iteration of the outer loop, *line* equals 1, so the inner loop executes once, printing the number 1 and a space. Then we print a newline character because line 19 is part of the outer *for* loop. The outer loop then sets the value of *line* to 2, and the inner loop starts again at 1 and executes two times (until *number* equals the line number in the outer loop). Then we again print a newline. This operation continues until the *line* exceeds 5, when the outer loop terminates. The output from Example 6.17 is shown in **Figure 6.26**.

Figure 6.26 Output from Example 6.17

Description

Note that we needed to use different names for our *for* loop control variables. The loop control variable *line* is in scope from lines 10 to 20, which includes the inner *for* loop.

Let’s look at another example of a nested loop. We’ll let the user enter positive integers, with a 0 being the sentinel value. For each number, we'll find all its factors; that is, we will find all the integers that are evenly divisible into the number, except 1 and the number itself.

If a number is evenly divisible by another, the remainder after division will be 0. The modulus operator (`%`) will be useful here, because it calculates the remainder after integer division. Thus, to find all the factors of a number, we can test all integers from 1 up to the number to see if the remainder after division is 0. But let's think about whether that's a good approach. The number 1 will be a factor for every number, because every number is evenly divisible by 1. So we can test integers beginning at 2. Then, because 2 is the smallest factor, there's no need to test integers higher than *number* / 2. Thus, our range of integers to test will be from 2 to *number* / 2.

For this example, we'll use a *for* loop nested inside a *while* loop. The pseudocode for this example is

```
read first number // priming read
while number is not 0
{
     print "The factors for number are "
     for factor = 2 to ( number / 2 ) by 1
     {
       if number % factor is 0
            print factor and a space
     }
     print a new line

     read next number // update read
}
```

Description

But what happens if we don't find any factors for a number? In that case, the number is a prime number. We can detect this condition by using a *boolean* flag variable. We set the flag to *false* before starting the *for* loop that checks for factors. Inside the *for* loop, we set the flag to *true* when we find a factor. In other words, we signal (or flag) the fact that we found a factor. Then, after the *for* loop terminates, we check the value of the flag. If it is still *false*, we did not find any factors and the number is prime. Our pseudocode for this program now becomes

```
read first number // priming read
while number is not 0
{
   print "The factors for number are "
   set flag to false
   for factor = 2 to ( number / 2 ) by 1
   {
       if number % factor is 0
       {
          print factor and a space
          set flag to true
       }
   }
   if flag is false
      print "number is prime"

   print a new line

   read next number // update read
}
```

Description

Since we want to read positive numbers only, the lines “read first number” and “read next number” in the preceding pseudocode will actually be more complex than a simple statement. Indeed, we will prompt the user to enter a positive number until the user does so. In order to do that, we will use a *do/while* loop to validate the input from the user. Therefore, inside the *while* loop, we nest not only a *for* loop, but also a *do/while* loop. In the interest of keeping the pseudocode simple, we did not show that *do/while* loop. However, it is included in the code in **Example 6.18** at lines 17–23 and 45–51.

Example 6.18 Finding Factors

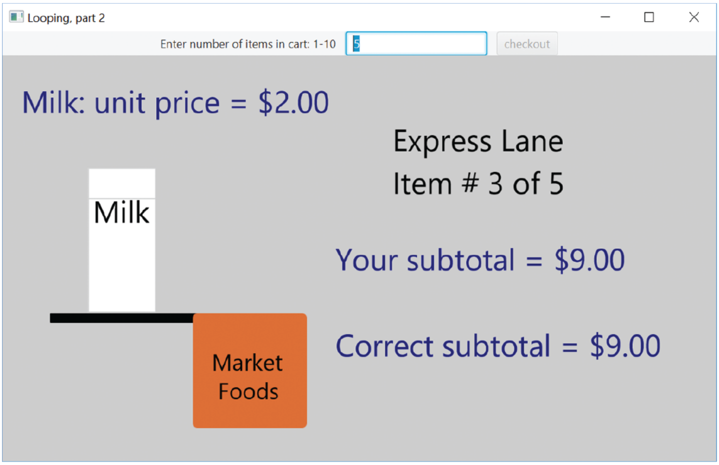
```
 1 /* Factors of integers
 2    with checks for primes
 3    Anderson, Franceschi
 4 */
 5 import java.util.Scanner;
 6
 7 public class Factors
 8 {
 9   public static void main( String [ ] args )
10   {
11     int number; // positive integer entered by user
12     final int SENTINEL = 0;
13     boolean factorsFound; // flag signals whether factors are foun
14
15     Scanner scan = new Scanner( System.in );
16
17     // priming read
18     do
19     {
20       System.out.print( "Enter a positive integer "
21                          + "or 0 to exit > " );
22       number = scan.nextInt( );
23     } while ( number < 0 );
24
25     while ( number != SENTINEL )
26     {
27       System.out.print( "Factors of " + number + ": " );
28       factorsFound = false; // reset flag to no factors
29
30       for ( int factor = 2; factor <= number / 2; factor++ )
31       {
32         if ( number % factor == 0 )
33         {
34            System.out.print( factor + " " );
35            factorsFound = true;
36         }
37       } // end of for loop
38
39       if ( ! factorsFound )
40           System.out.print( "none, " + number + " is prime" );
41
42       System.out.println( ); // print a newline
43       System.out.println( ); // skip a line
44
45       // read next number
46       do
47       {
48          System.out.print( "Enter a positive integer "
49                             + "or 0 to exit > " );
50          number = scan.nextInt( );
51       } while ( number < 0 );
52     } // end of while loop
53   }
54 }
```

Description

Translating this pseudocode into Java, we get the code shown in Example 6.18; the output of a sample run of the program is shown in **Figure 6.27**.

Figure 6.27 Output of Finding Factors

Description



```
Milk: unit price = $2.00

Express Lane
Item # 3 of 5

Your subtotal = $9.00

Correct subtotal = $9.00
```

6.12 Programming Activity 2: Using *for* Loops

In this activity, you will write a *for* loop:

> For this Programming Activity, you will again calculate the total cost of the items in a grocery cart. This time, however, you will write the program for the Express Lane. In this lane, the customer is allowed up to 10 items. The user will be asked for the number of items in the grocery cart. Your job is to write a *for* loop to calculate the total cost of the items in the cart.

Like Programming Activity 1, the framework will animate your *for* loop, displaying the items in the cart moving down a conveyor belt toward a cashier station (a grocery bag). It will also display the unit price of the item, the correct subtotal, and your current subtotal. By comparing the correct subtotal to your subtotal, you will be able to check whether your code is calculating the correct value.

Figure 6.28 demonstrates the animation. The cart contains five items. The third item, a carton of milk, is being scanned at a unit price of $2.00, bringing the correct subtotal for the cart to $9.00.

Figure 6.28 Sample Animation

Description

Instructions

Copy the files in this chapter’s Programming Activity 2 folder in the source code provided with this text to a folder on your computer. Searching for five stars (*****) in the *CashierDrawing.java* code will show you where to add your code. You will add your code inside the *checkout* method of the *CashierDrawing* class (the method header for the *checkout* method has already been coded for you). **Example 6.19** shows a fragment of the *CashierDrawing* class, where you will add your code:

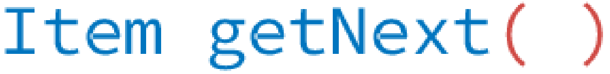
Example 6.19 The *checkout* Method in the *CashierDrawing* Class

```
public void checkout( int numberOfItems )
{
   /* ***** Student writes the body of this method ***** */
   //
   //  The parameter of this method, numberOfItems,
   //  represents the number of items in the cart. The
   //  user will be prompted for this number.
   //
   //  Using a for loop, calculate the total price
   //  of the groceries in the cart.
   //
   //  The getNext method (in this CashierDrawing class) returns the next
   //  item in the cart, which is an Item object (we do not
   //  know which item will be returned; this is randomly generated).
   //  getNext does not take any arguments. Its API is
   //      Item getNext( )
   //
   // As the last statement of the body of your for loop,
   // you should call the animate method.
   // The animate method takes one argument a double,
   // which is your current subtotal.
   // For example, if the name of your variable representing
   // the current subtotal is total, your call to the animate
   // method should be:
   //     animate( total );
   //
   // The getPrice method of the Item class
   // returns the price of the Item object as a double.
   // The getPrice method does not take any arguments. Its API is
   // double getPrice( )
   //
   //  End of student code
   //
   //
}
```

Description

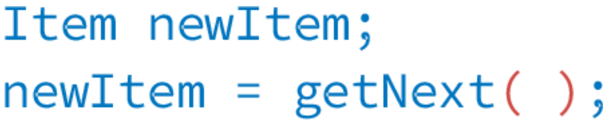
To write the body of your *for* loop, you can use the following methods:

- You can access items in the cart using the *getNext* method of the *CashierDrawing* class, which has the following API:



```
Item getNext( )
```

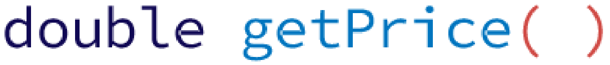
The *getNext* method returns an *Item* object, which represents an *Item* in the cart. As you can see, the *getNext* method does not take any arguments. Since we call the method *getNext* from inside the *CashierDrawing* class, we can simply call the method without an object reference. For example, a call to *getNext* could look like the following:



```
Item newItem;
newItem = getNext( );
```

Description

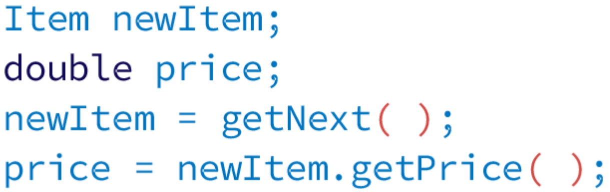
- After you get a new *Item*, you can “scan” the item to get its price by calling the *getPrice* method of the *Item* class. The *getPrice* method has this API:



```
double getPrice( )
```

Description

Thus, you would get the next item, then get its price using code like the following:



```
Item newItem;
double price;
newItem = getNext( );
price = newItem.getPrice( );
```

Description

When you have finished writing the code for the *checkout* method, compile *CashierDrawing.java* and run the application from the *CashierApplication* class. When the application finishes executing, verify that your code is correct by:

- checking that your subtotal matches the correct subtotal displayed
- checking that you have processed all the items in the cart by verifying that the current item number matches the total number of items. For example, if the cart has five items, check that the message in the top right of the screen displays: `Item # 5 of 5`.

Troubleshooting

If your method implementation does not animate or animates incorrectly, check these items:

- Verify that you have correctly coded the header of your *for* loop.
- Verify that you have correctly coded the body of the loop.

DISCUSSION QUESTIONS

1. Explain why a *for* loop is appropriate for this activity.
2. Explain how you set up your *for* loop; that is, what initialization statement did you use, what was your condition, and what was the loop update statement?

© Lois GoBe/Shutterstock.

CHAPTER REVIEW

6.13 Chapter Summary

- Looping repeats a set of operations for each input item while a condition is *true*.
- The *while* loop is especially useful for event-controlled looping. The *while* loop executes a set of operations in the loop body as long as the loop condition is *true*. Each execution of the loop body is an iteration of the loop.
- If the loop condition evaluates to *false* the first time it is evaluated, the body of the *while* loop is never executed.
- If the loop condition never evaluates to *false*, the result is an infinite loop.
- In event-controlled looping, processing of items continues until the end of input is signaled either by a sentinel value or by reaching the end of the file.
- A sentinel value is a special input value that signals the end of the items to be processed. With a sentinel value, we perform a priming read before the *while* loop. The body of the loop processes the input, then performs an update read of the next data item.
- When reading data from an input file, we can test whether we have reached the end of the file by calling a *hasNext* method of the *Scanner* class.
- In the accumulation programming technique, we initialize a

total variable to 0 before starting the loop. In the loop body, we add each input value to the total. When the loop completes, the current total is the total for all processed input values.

- In the counting programming technique, we initialize a count variable to 0 before starting the loop. In the loop body, we increment the count variable for each input value that meets our criteria. When the loop completes, the count variable contains the number of items that met our criteria.
- To find an average, we combine accumulation and counting. We add input values to the total and increment the count. When the loop completes, we calculate the average by dividing the total by the count. Before computing the average, however, we should verify that the divisor (that is, the count) is not 0.
- To find the maximum or minimum values in a set of input, we assign the first input to a running maximum or minimum. In the loop body, we compare each input value to our running maximum or minimum. If the input value is less than the running minimum, we assign the input value to the running minimum. Similarly, if the input value is greater than the running maximum, we assign the input value to the running maximum. When the loop completes, the running value is the maximum or minimum value of all the input values.
- To avoid generating exceptions when the user types characters other than the data type expected, use the *hasNext* methods of the *Scanner* class.
- To construct a loop condition, construct the inverse of the loop termination condition.
- When testing a program that contains a loop, test that the program produces correct results by inputting values and comparing the results with manual calculations. Also test that the results are correct if the loop body never executes. Finally, test the results with input that is invalid.
- The *do/while* loop checks the loop condition after executing the loop body. Thus, the body of a *do/while* loop always executes at least once. This type of loop is useful for

validating input.

- The *for* loop is useful for count-controlled loops, that is, loops for which the number of iterations is known when the loop begins.
- When the *for* loop is encountered, the initialization statement is executed. Then the loop condition is evaluated. If the condition is *true*, the loop body is executed. The loop update statement is then executed and the loop condition is reevaluated. Again, if the condition is *true*, the loop body is executed, followed by the loop update, then the reevaluation of the condition, and so on, until the condition evaluates to *false*.
- Typically, we use a loop control variable in a *for* loop. We set its initial value in the initialization statement, increment or decrement its value in the loop update statement, and check its value in the loop condition.
- The loop update statement can increment or decrement the loop variable by any value.
- In a *for* loop, it is important to test that the starting and ending values of the loop variable are correct. Also test with input for which the *for* loop body does not execute at all.

6.14 Exercises, Problems, and Projects

6.14.1 Multiple Choice Exercises

1. How do you discover that you have an infinite loop in your code?

    - The code does not compile.
    - The code compiles and runs but gives the wrong result.
    - The code runs forever.
    - The code compiles, but there is a run-time error.

6. If you want to execute a loop body at least once, what type of loop would you use?

    - *for* loop
    - *while* loop
    - *do/while* loop
    - none of the above

11. What best describes a *for* loop?

    - It is a count-controlled loop.
    - It is an event-controlled loop.
    - It is a sentinel-controlled loop.

15. You can simulate a *for* loop with a *while* loop.

    - true
    - false

6.14.2 Reading and Understanding Code

1. What is the output of this code sequence? (The user successively enters 3, 5, and –1.)

```
System.out.print( "Enter an int > " );
int i = scan.nextInt( );
while ( i != -1 )
{
   System.out.println( "Hello" );

   System.out.print( "Enter an int > " );
   i = scan.nextInt( );
}
```

   Description

2. What is the output of this code sequence? (The user successively enters 3, 5, and –1.)

```
int i = 0;
while ( i != -1 )
{
   System.out.println( "Hello" );
   System.out.print( "Enter an int > " );
   i = scan.nextInt( );
}
```

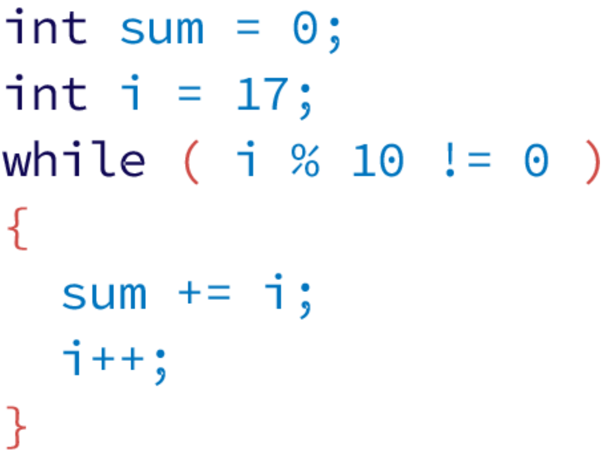
   Description

3. What is the output of this code sequence? (The user successively enters 3, 5, and –1.)

```
System.out.print( "Enter an int > " );
int i = scan.nextInt( );
while ( i != -1 )
{
   System.out.print( "Enter an int > " );
   i = scan.nextInt( );

   System.out.println( "Hello" );
}
```

   Description

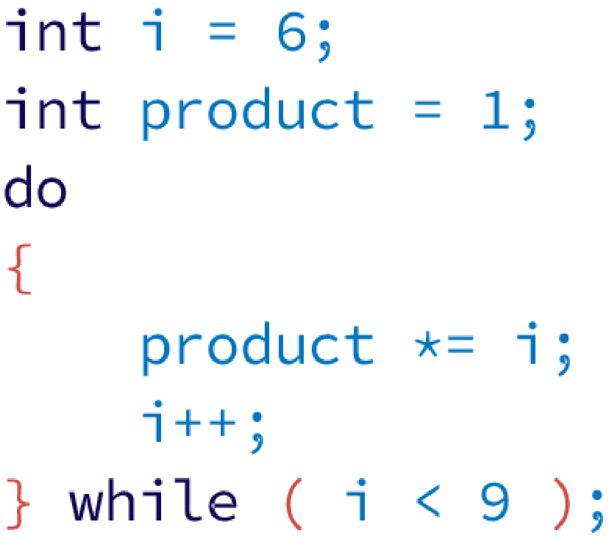
4. What are the values of *i* and *sum* after this code sequence is executed?



```
int sum = 0;
int i = 17;
while ( i % 10 != 0 )
{
   sum += i;
   i++;
}
```

Description

5. What are the values of *i* and *product* after this code sequence is executed?

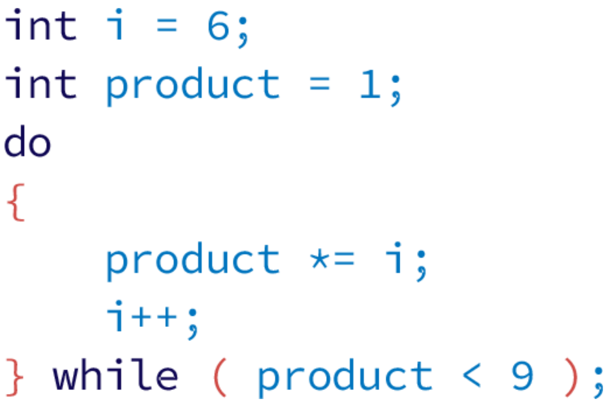


```
int i = 6;
int product = 1;
do
{
     product *= i;
     i++;
} while ( i < 9 );
```

Description

6. What are the values of *i* and *product* after this code

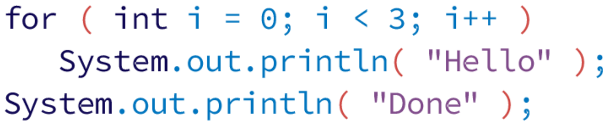
sequence is executed?



```
int i = 6;
int product = 1;
do
{
     product *= i;
     i++;
} while ( product < 9 );
```

Description

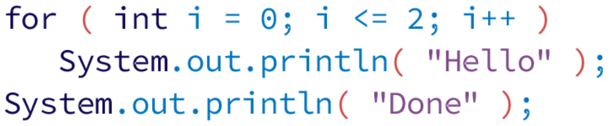
7. What is the output of this code sequence?



```
for ( int i = 0; i < 3; i++ )
   System.out.println( "Hello" );
System.out.println( "Done" );
```

Description

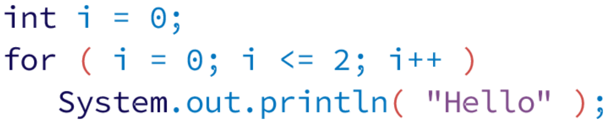
8. What is the output of this code sequence?



```
for ( int i = 0; i <= 2; i++ )
   System.out.println( "Hello" );
System.out.println( "Done" );
```

Description

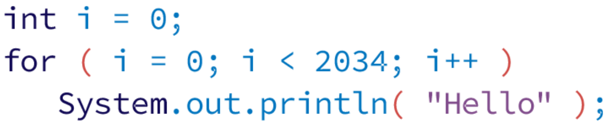
9. What is the value of *i* after this code sequence is executed?



```
int i = 0;
for ( i = 0; i <= 2; i++ )
   System.out.println( "Hello" );
```

Description

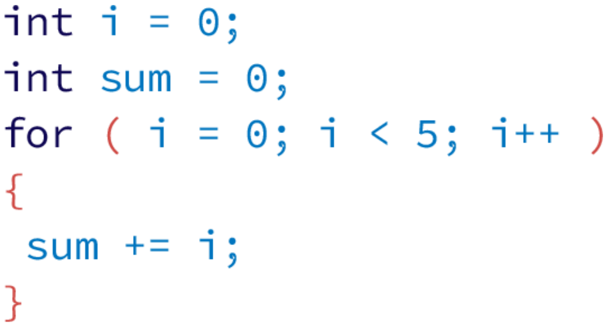
10. What is the value of *i* after this code sequence is executed?



```
int i = 0;
for ( i = 0; i < 2034; i++ )
   System.out.println( "Hello" );
```

Description

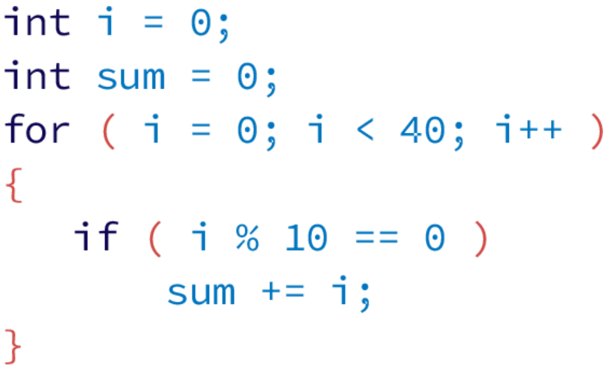
11. What are the values of *i* and *sum* after this code sequence is executed?



```
int i = 0;
int sum = 0;
for ( i = 0; i < 5; i++ )
{
 sum += i;
}
```

Description

12. What are the values of *i* and *sum* after this code sequence is executed?



```
int i = 0;
int sum = 0;
for ( i = 0; i < 40; i++ )
{
    if ( i % 10 == 0 )
          sum += i;
}
```

Description

13. What is the value of *sum* after this code sequence is executed?

```
int sum = 0;
for ( int i = 1; i < 10; i++ )
{
     i++;
     sum += i;
}
```

Description

14. What is the value of *sum* after this code sequence is executed?

```
int sum = 0;
for ( int i = 10; i > 5; i-- )
{
     sum += i;
}
```

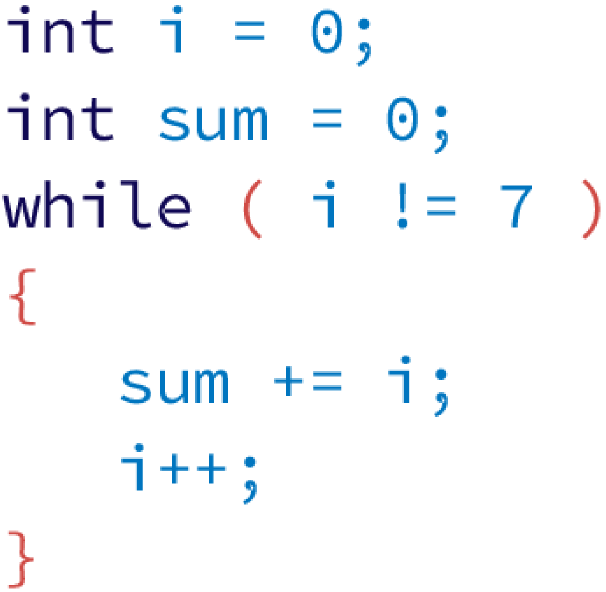
Description

15. What is printed when this code sequence is executed?

```
for ( int i = 0; i < 5; i++ )
{
   System.out.println( Math.max( i, 3 ) );
}
```

Description

16. What are the values of *i* and *sum* after this code sequence is executed?

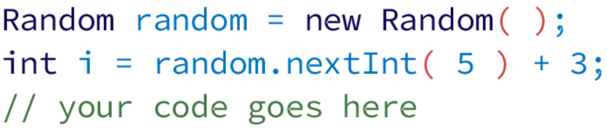


```
int i = 0;
int sum = 0;
while ( i != 7 )
{
     sum += i;
     i++;
}
```

Description

6.14.3 Fill In the Code

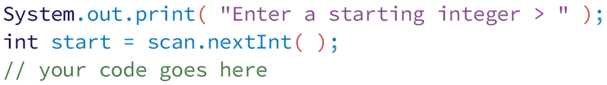
1. This *while* loop generates random integers between 3 and 7 until a 5 is generated and prints all the random integers, excluding 5.

   ```
   Random random = new Random( );
   int i = random.nextInt( 5 ) + 3;
   // your code goes here
   ```

   Description

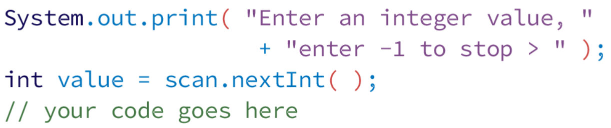
2. This *while* loop takes an integer input from the user, then prompts for additional integers and prints all integers that are greater than or equal to the original input until the user enters 20, which is not printed.

   ```
   System.out.print( "Enter a starting integer > " );
   int start = scan.nextInt( );
   // your code goes here
   ```

Description

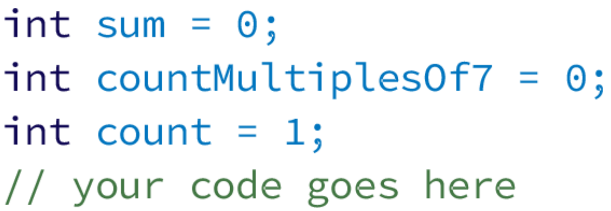
3. This *while* loop takes integer values as input from the user and finds the sum of those integers until the user types in the value –1 (which is not added).



```
System.out.print( "Enter an integer value, "
                   + "enter -1 to stop > " );
int value = scan.nextInt( );
// your code goes here
```

Description

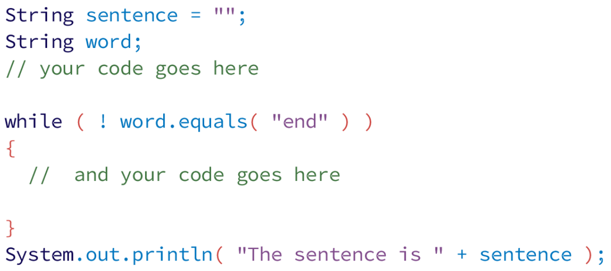
4. This loop calculates the sum of the first four positive multiples of 7 using a *while* loop (the sum will be equal to 7 + 14 + 21 + 28 = 70).



```
int sum = 0;
int countMultiplesOf7 = 0;
int count = 1;
// your code goes here
```

Description

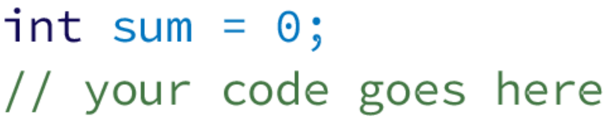
5. This loop takes words as input from the user and concatenates them until the user types in the word “end” (which is not concatenated). The code then outputs the concatenated *String*.



```
String sentence = "";
String word;
// your code goes here

while ( ! word.equals( "end" ) )
{
  //  and your code goes here

}
System.out.println( "The sentence is " + sentence );
```

Description

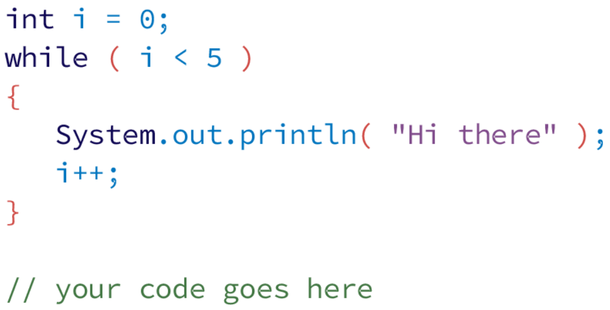
6. This loop reads integers from a file (already associated with the *Scanner* object reference *file*) and computes the sum. We don’t know how many integers are in the file.



```
int sum = 0;
// your code goes here
```

Description

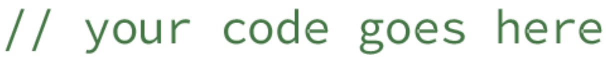
7. Here is a *while* loop; write the equivalent *for* loop.



```
int i = 0;
while ( i < 5 )
{
   System.out.println( "Hi there" );
   i++;
}

// your code goes here
```

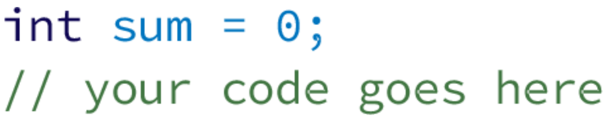
Description

8. This loop reads integers from the user until the user enters either 0 or 100. Then it prints the sum of the numbers entered (excluding the 0 or 100).



```
// your code goes here
```

9. This loop calculates the sum of the integers from 1 to 5 using a *for* loop.



```
int sum = 0;
// your code goes here
```

Description

6.14.4 Identifying Errors in Code

1. Where is the problem with this code sequence (although this code sequence does compile)?

   ```
   int i = 0;
   while ( i < 3 )
      System.out.println( "Hello" );
   ```

   Description

2. Where is the error in this code sequence that is supposed to read and echo integers until the user enters –1?

   ```
   int num;
   Scanner scan = new Scanner( System.in );
   while ( num != -1 )
   {
      System.out.print( "Enter an integer > " );
      num = scan.nextInt( );
      System.out.println( num );
   }
   ```

   Description

3. The following code sequence intends to print *Hello* three times; however, it prints *Hello* only once. Where is the problem in this code sequence?

   ```
   for ( int i = 0; i < 3; i++ );
      System.out.println( "Hello" );
   ```

   Description

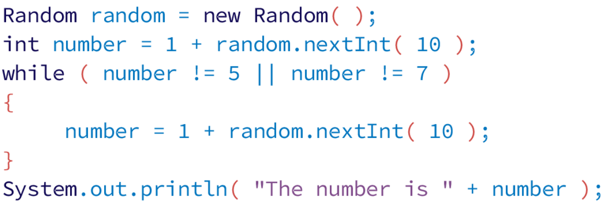
4. Where is the error in this code sequence, which is intended to print *Hello* 10 times?

   ```
   for ( int i = 10; i > 0; i++ )
      System.out.println( "Hello" );
   ```

   Description

5. Where is the problem with this code sequence? The code is intended to generate random numbers between 1 and 10

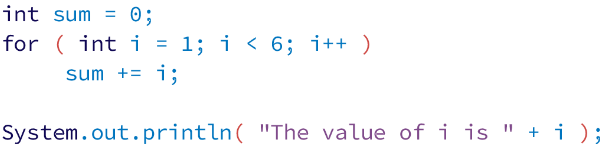
until the number is either a 7 or a 5.



```
Random random = new Random( );
int number = 1 + random.nextInt( 10 );
while ( number != 5 || number != 7 )
{
     number = 1 + random.nextInt( 10 );
}
System.out.println( "The number is " + number );
```

Description

6. Where is the error with this code sequence?

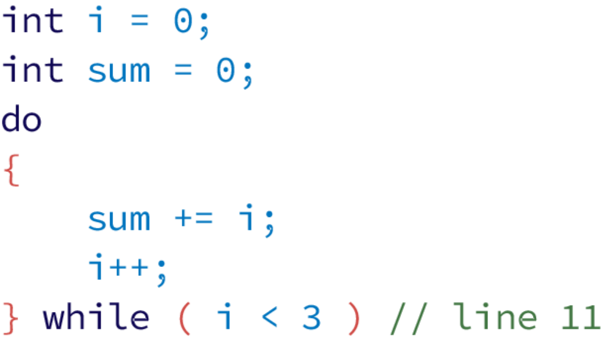


```
int sum = 0;
for ( int i = 1; i < 6; i++ )
     sum += i;

System.out.println( "The value of i is " + i );
```

Description

6.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. You coded the following in the class *Test.java*:



```
int i = 0;
int sum = 0;
do
{
     sum += i;
     i++;
} while ( i < 3 ) // line 11
```

Description
At compile time, you get the following error:

```
Test.java:11: error: ';' expected
while( i < 3 ) // line 11
              ^
1 error
```

Description

Explain what the problem is and how to fix it.

2. You coded the following in the class *Test.java*:

```
int i = 0;
while ( i < 3 )
{
   System.out.println( "Hello" );
   i--;
}
```

Description

The code compiles but never terminates.
Explain what the problem is and how to fix it.

3. You coded the following in the class *Test.java*:

```
for ( int i = 0; i++; i < 3 ) // line 5
   System.out.println( "Hello" );
```

Description

At compile time, you get the following error:

```
Test.java:5: error: not a statement
for ( int i = 0; i++; i < 3 ) // line 5
                        ^
1 error
```

Description

Explain what the problem is and how to fix it.

4. You coded the following in the class *Test.java*:

```
for ( int i = 1; i < 3; i++ ) // line 5
   System.out.println( "Hello" );
```

Description

The code compiles and runs, but prints *Hello* only twice, whereas we expected to print *Hello* three times.

Explain what the problem is and how to fix it.

5. You coded the following in the class *Test.java*:

```
int product = 1;
for ( int i = 1, i < 5, i++ ) // line 8
   product *= i;
System.out.println( "Product is " + product );    // line 10
```

Description

At compile time, you get the following errors:

```
Test.java:8: error: ';' expected
    for ( int i = 1, i < 5, i++ ) // line 8
                      ^
Test.java:8: error: illegal start of type
    for ( int i = 1, i < 5, i++ )       // line 8
                         ^
2 errors
```

Description

Explain what the problem is and how to fix it.

6. You coded the following in the class *Test.java*:

```
for ( int i = 0; i < 3; i++ )
   System.out.println( "Hello" );
System.out.println( "i = " + i ); // line 8
```

Description

At compile time, you get the following error:

```
Test.java:8: error: cannot find symbol
       System.out.println( "i = " + i ); // line 8
                                    ^
symbol : variable i
location: class Test
1 error
```

Description

Explain what the problem is and how to fix it.

7. You coded the following in the class *Test.java*:

```
int i = 0;
for ( int i = 0; i < 3; i++ ) // line 6
   System.out.println( "Hello" );
```

Description

At compile time, you get the following error:

```
Test.java:6: error: variable i is already defined in method main( String[] )
for( int i = 0; i < 3; i++ )      // line 6
         ^
1 error
```

Description

Explain what the problem is and how to fix it.

6.14.6 Write a Short Program

1. Write a program that prompts the user for a value greater than 10 as an input (you should loop until the user enters a valid value) and finds the square root of that number and the square root of the result, and continues to find the square root of the result until you reach a number that is smaller than 1.01. The program should output how many times the square root operation was performed.
2. Write a program that expects a word containing the @ character as an input. If the word does not contain an @ character, then your program should keep prompting the user for a word. When the user types in a word containing an @ character, the program should simply print the word and terminate.
3. Write a program that reads *double* values from a file named *input.txt* and outputs the average.
4. Write a program that uses a *for* loop to output the sum of all the integers between 10 and 20, inclusive, that is, 10 + 11 + 12 + ... + 19 + 20.
5. Write a program that uses a *for* loop to output the product

of all the integers between 3 and 7, inclusive, that is, 3 * 4 * 5 * 6 * 7.

6. Write a program that uses a *for* loop to count how many multiples of 7 are between 33 and 97, inclusive.
7. Write a program that reads a value (say *n*) from the user and outputs *Hello World n* times. Verify that the user has entered an integer. If the input is 3, the output will be *Hello World* printed three times.
8. Write a program that takes a word as an input from the keyboard and outputs each character in the word, separated by a space.
9. Write a program that takes a value as an input from the keyboard and outputs the factorial of that number; the factorial of an integer *n* is *n* * (*n* – 1) * (*n* – 2) * … * 3 * 2 * 1. For instance, the factorial of 4 is 4 * 3 * 2 * 1, or 24.
10. Using a loop, write a program that takes 10 integer values from the keyboard and outputs the minimum value of all the values entered.
11. Alter Example 6.14 that counts the tokens in a sentence so that it prints each token and counts the number of tokens.

6.14.7 Programming Projects

1. Write a program that inputs a word representing a binary number (0s and 1s). First, your program should verify that it is indeed a binary number, that is, the number contains only 0s and 1s. If that is not the case, your program should print a message that the number is not a valid binary number. Then, your program should count how many 1s are in that word and output the count.

2. Perform the same operations as Question 54, with the following modification: If the word does not represent a valid binary number, the program should keep prompting the user for a new word until a word representing a valid binary number is input by the user.
3. Write a program that inputs a word representing a binary number (0s and 1s). First, your program should check that it is indeed a binary number, that is, the number contains only 0s and 1s. If that is not the case, your program should output that the number is not a valid binary number. If that word contains exactly two 1s, your program should output that that word is “accepted,” otherwise that it is “rejected.”
4. Perform the same operations as Question 56, with the following modification: If the word does not represent a valid binary number, the program should keep prompting the user for a new word until a word representing a valid binary number is input by the user.
5. Write a program that inputs a word representing a binary number (0s and 1s). First, your program should check that it is indeed a binary number, that is, that it contains only 0s and 1s. If that is not the case, your program should output that the number is not a valid binary number. If that word contains at least three consecutive 1s, your program should output that that word is “accepted,” otherwise that it is “rejected.”
6. Write a program that inputs 7 double values from a file *dja.txt* that represent the Dow Jones Average for 7 days. Your program should output the lowest value for those 7 days and the number of the day on which the lowest value occurred. For this program, instead of setting the initial minimum value to the first value in the file, use the

maximum value for a *double*. The Java Class Library provides this value as a constant in the *Double* wrapper class. Be sure to handle the case of the file being empty.

7. Write a program that takes website names as keyboard input until the user types the word *stop* and counts how many of the website names are commercial website names (i.e., end with *.com*), then outputs that count.
8. Using a loop, write a program that takes 10 values representing exam grades (between 0 and 100) from the keyboard and outputs the minimum value, maximum value, and average value of all the values entered. Your program should not accept values less than 0 or greater than 100.
9. Write a program that takes an email address as an input from the keyboard and, using a loop, steps through every character looking for an @ sign. If the email address has exactly one @ character, then print a message that the email address is valid; otherwise, print a message that it is

invalid.

10. Write a program that takes a user ID as an input from the keyboard and steps through every character, counting how many digits are in the user ID; if there are exactly two digits, output that the user ID is valid, otherwise that it is invalid.
11. Write a program that takes an integer value as an input and converts that value to its binary representation; for instance, if the user inputs 17, then the output will be 10001.
12. Write a program that takes a word representing a binary number (0s and 1s) as an input and converts it to its decimal representation; for instance, if the user inputs 101, then the output will be 5; you can assume that the *String* is guaranteed to contain only 0s and 1s.
13. Write a program that simulates an XOR operation. The input should be a word representing a binary number (0s and 1s). Your program should XOR all the digits from left to right and output the results as "True" or "False." In an XOR operation, *a* XOR *b* is true if *a* or *b* is *true* but not both; otherwise, it is *false*.

    In this program, we will consider the character "1" to represent true and a "0" to represent false. For instance, if the input is 1011, then the output will be 1 (1 XOR 0 is 1, then 1 XOR 1 is 0, then 0 XOR 1 is 1, which causes the output to be "True"). You can assume that the input word is guaranteed to contain only 0s and 1s.
14. Write a program that takes a sentence as an input and checks whether that sentence is a palindrome. A palindrome is a word, phrase, or sentence that is symmetrical; that is, it is spelled the same forward and backward. Examples are "otto," "mom," and "Able was I ere

I saw Elba.” Your program should be case insensitive; that is, “Otto” should also be counted as a palindrome.

15. Write a program that takes an HTML-like sequence as an input and checks whether that sequence has the same number of opening brackets (<) and closing brackets (>).

6.14.8 Technical Writing

1. In programming, a programmer can make syntax errors that lead to a compiler error; these errors can then be corrected. Other errors can lead to a run-time error; these errors can also be corrected. Logic errors, however, can lead to an incorrect result or no result at all. Discuss examples of logic errors that can be made when coding loops and the consequences of these logic errors.
2. Discuss how you would detect whether you have an infinite loop in your code.

6.14.9 Group Project (for a group of 1, 2, or 3 students)

1. Often on a webpage, the user is asked to supply personal information, such as a telephone number. Your program should take an input from the keyboard representing a telephone number. We will consider that the input is a valid telephone number if it contains exactly 10 digits and any number of dash (-) and whitespace characters. Keep prompting the user for a telephone number until the user gives you a valid one. Once you have a valid telephone number, you should assume that the digits (only the digits, not the hyphen[s] nor the white space) in the telephone number may have been encrypted by shifting each number by a constant value. For instance, if the shift is 2, a 0

becomes a 2, a 1 becomes a 3, a 2 becomes a 4, …, an 8 becomes a 0, and a 9 becomes a 1. However, we know that the user is from New York where the decrypted area code (after the shift is applied), represented by the first three digits of the input, is 212. Your program needs to decrypt the telephone number and output the decrypted telephone number with the format 212-xxx-xxxx, as well as the shift value of the encryption. If there was an error in the input and the area code cannot be decrypted to 212, you should output that information.

© Lois GoBe/Shutterstock.

CHAPTER 7

Object-Oriented Programming, Part 2: User-Defined Classes

CHAPTER CONTENTS

Introduction
7.1 Defining a Class
7.2 Defining Instance Variables
7.3 Writing Class Methods
7.4 Writing Constructors
7.5 Writing Accessor Methods
7.6 Writing Mutator Methods
7.7 Writing Data Manipulation Methods
7.8 **Programming Activity 1: Writing a Class Definition, Part 1**
7.9 The Object Reference *this*
7.10 The *toString* and *equals* Methods
7.11 *Static* Class Members
7.12 Another Example: The *Rational* Class
7.13 Graphical Objects
7.14 Enumeration Types
7.15 **Programming Activity 2: Writing a Class Definition, Part 2**

7.16 Creating Packages
7.17 Generating Web-Style Documentation with Javadoc
7.18 Chapter Summary
7.19 Exercises, Problems, and Projects

- 7.19.1 Multiple Choice Exercises
- 7.19.2 Reading and Understanding Code
- 7.19.3 Fill In the Code
- 7.19.4 Identifying Errors in Code
- 7.19.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
- 7.19.6 Write a Short Program
- 7.19.7 Programming Projects
- 7.19.8 Technical Writing
- 7.19.9 Group Project

Introduction

When you see the title of this chapter, you might say, “Finally, we get to write our own classes.” Actually, we’ve been writing classes all along. All Java source code belongs to a class. The classes we’ve been writing are application classes. Now it’s time to write some service classes—classes that encapsulate data and methods for use by applications or service classes. These are called **user-defined classes** because we, rather than the Java authors, create them.

First, let’s take a moment to examine why we want to create user-defined classes.

We have written a lot of programs using Java’s primitive data types (*boolean, char, int, double*, etc.), but the real world requires manipulation of more complex data than just individual *booleans* or *ints*. For example, if you are the programmer for an online bookstore, you will need to manipulate data associated with books. Books typically have an ISBN, a title, an author, a price, an in-stock quantity, and perhaps other pieces of data. We can create a *Book* class so that each object will hold the data for one book. For example, the ISBN, the title, and the author can be represented by *Strings*, the price by a *double*, and the in-stock quantity by an *int*. If we create this *Book* class, our program will be able to store and manipulate all the data of a book as a whole. This is one of the concepts of object-oriented programming.

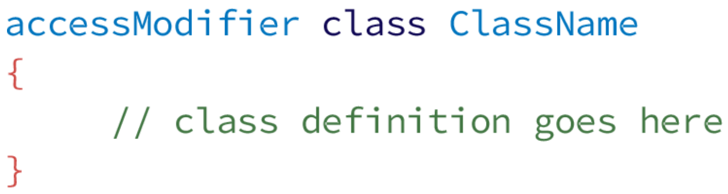
By incorporating into the class the methods that work with the book data, we also are able to hide the details involved with handling that data. An application can simply call the methods as needed. Thus, creating your own classes can simplify your program.

Finally, a well-written class can be reused in other programs. Thus, user-defined classes speed up development.

7.1 Defining a Class

Classes encapsulate the data and functionality for a person, place, or thing, or more generally, an object. For example, a class might be defined to represent a student, a college, or a course.

To define a class, we use the following syntax:



```
accessModifier class ClassName
{
     // class definition goes here
}
```

Description

SOFTWARE ENGINEERING TIP

Use a noun for the class name, and start the class name with a capital letter.

This syntax should look familiar as the first line in our programs. You may also notice that the class names in the Java Class Library are nouns and start with a capital letter: *Scanner, String, Math*, for example. The Java developers encourage programmers to use these conventions to name their user-defined classes.

Inside the curly braces we define the data of the class, called its **fields**, and the methods. An important function performed by the class methods is maintaining the values of the class data for the **client programs**, which are the users of the class, in that the clients create objects and call the methods of the class. Our programs have been clients of many Java classes, such as *String, DecimalFormat*, and *Math*. The fields and methods of a class are called the **members** of the class.

For each class and for each member of a class, we need to provide an **access modifier** that specifies where the class or member can be used (see **Table 7.1**). The possible access modifiers are *public, private*, and *protected*, or no modifier at all, which results in package access. The *public* access modifier allows the class or member to be used, or **referenced**, by methods of the same or other classes. The *private* access modifier specifies that the class or member can be referenced only by methods of the same class. Package access specifies that the class or member can be accessed by methods in classes that are in the same package or in the same folder. Later in the chapter, we will learn how to create our own package.

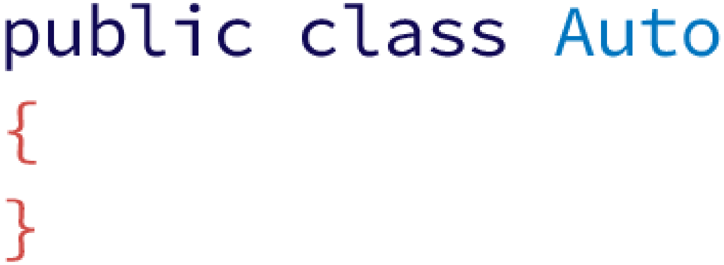
Access Modifiers

Access Modifier	Class or Member Can Be Referenced By ...
public	methods of the same class, as well as methods of other classes
private	methods of the same class only
protected	methods in the same class, as well as methods of subclasses and methods in classes in the same package
no modifier (package access)	methods in the same package or same folder only

Description

Typically, the *accessModifier* for a class will be *public*, and we know that a *public* class must be stored in a file named *ClassName.java* where *ClassName* is the name of the class.

Let's start to define a class that represents an automobile, which we can use to calculate miles per gallon. We'll name the class *Auto*, and we'll use the *public* access modifier so that any application can use this class. The class header will look like the following:



```
public class Auto
{
}
```

Description

When we write a class, we will make known the *public* method names and their APIs so that a client program will know how to instantiate objects and call the methods of the class. We will not publish the implementation (or code) of the class, however. In other words, we will publish the APIs of the methods, but not the method bodies. This is called **data hiding.** A client program can use the class without knowing how the class is implemented, and we, as class authors, can change the implementation of the methods as long as we don't change the interface, or APIs.

7.2 Defining Instance Variables

The instance variables of a class hold the data for each object of that class. Thus, we also say that the instance variables represent the properties of the object. Each object, or instance of a class, gets its own copy of the instance variables, each of which can be given a value appropriate to that object. The values of the instance variables, therefore, can represent the state of the object.

Instance variables are defined using the following syntax:

```
accessModifier dataType identifierList;
```

Description

The *private* modifier is typically used for the nonconstant instance variables of the class. This permits only methods of the same class to set or change the values of the instance variables. In this way, we achieve encapsulation; the class provides a protective shell around the data.

SOFTWARE ENGINEERING TIP

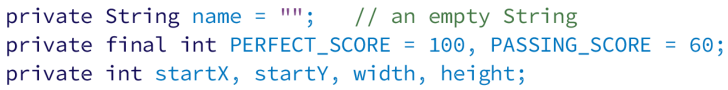
Define instance variables of a class as *private* so that only methods of the class will be able to set or change their values.

The data type of an instance variable can be any of Java's primitive types or a class type.

The *identifierList* consists of one or more names for instance variables of the same data type and can optionally assign initial values to the instance variables. If more than one instance variable name is given, a comma is used as a separator. By convention, identifier names for instance variables are nouns and begin with a lowercase letter; internal words begin with a capital letter. Each instance variable and class variable must be given a name that is unique to the class. It is legal to use the same names for instance variables in different classes, but within a class, the same name cannot be used for more than one instance variable or class variable. Thus, we say that the fields of a class have **class scope**.

Optionally, you can declare an instance variable to be a constant (*final*).

The following statements are examples of instance variable definitions:



```
private String name = "";    // an empty String
private final int PERFECT_SCORE = 100, PASSING_SCORE = 60;
private int startX, startY, width, height;
```

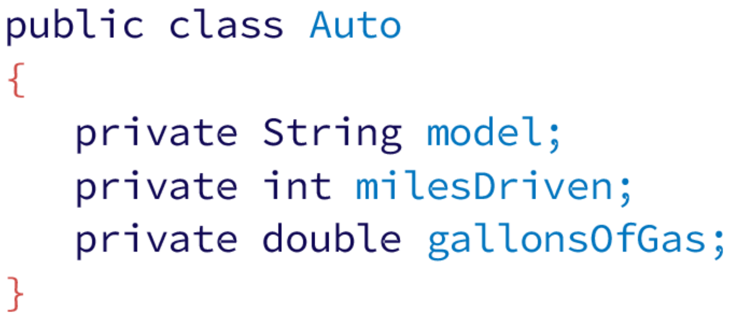
Description

What criteria should you use to select the instance variables of the class? The answer is to select the data that all objects will have in common. For example, for a *Student* class, you might select the student name, grade point average, and projected graduation date. For a *Calculator* class, you might select two operands, an operator, and a result.

SOFTWARE ENGINEERING TIP

Use nouns for identifier names for instance variables. Begin the identifier with a lowercase letter and capitalize internal words.

Thus, for our *Auto* class, we will define instance variables to hold the model of the automobile, the number of miles the auto has been driven, and the gallons of gas used. As a result, our *Auto* class definition now becomes the following:



```
public class Auto
{
   private String model;
   private int milesDriven;
   private double gallonsOfGas;
}
```

Description

SOFTWARE ENGINEERING TIP

Define instance variables for the data that all objects will have in common.

7.3 Writing Class Methods

We declared the instance variables of the *Auto* class as *private* so that only the methods of the *Auto* class will be able to access or change the values of the instance variables directly. Clients of the *Auto* class will need to use the methods of the class to access or change any of the instance variables. We will therefore need to write some methods.

SOFTWARE ENGINEERING TIP

Use verbs for method names. Begin the method name with a lowercase letter and begin internal words with a capital letter.

Methods have this syntax:

```
accessModifier returnType methodName( parameter list ) // method header
{
      // method body
}
```

Description

where *parameter list* is a comma-separated list of data types and variable names.

The method header syntax should be familiar because we've seen the API for many class methods. One difference is just a matter of semantics. The method caller sends **arguments**, or **actual parameters**, to the method; the method refers to these arguments as its **formal parameters**.

Because methods provide a function for the class, typically method names are verbs. Like instance variables, the method name should begin with a lowercase letter, with internal words beginning with a capital letter.

The access modifier for methods that provide services to the client will be *public*. Methods that provide services only to other methods of the class are typically declared to be *private*.

SOFTWARE ENGINEERING TIP

Declare methods that provide services to clients as *public*. Methods that will be called only by other methods of the class should be declared *private.*

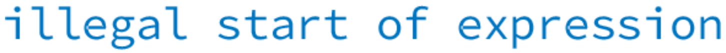
The return type of a method is the data type of the value that the method returns to the caller. The return type can be any of Java's primitive data types, any class type, or *void*. Methods with a return type of *void* do not return a value to the caller.

The body of each method, which consists of the code that performs the method's function, is written between the beginning and ending curly braces. Unlike *if* statements and loops, however, these curly braces are not optional; the curly braces are required, regardless of the number of statements in the method body.

COMMON ERROR TRAP

Forgetting to enclose the body of a method in curly braces generates one or more compiler errors.

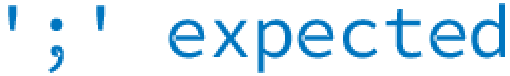
Several compiler errors can result from forgetting one or both of the curly braces. You might receive either of these messages:



```
illegal start of expression
```

Description

or



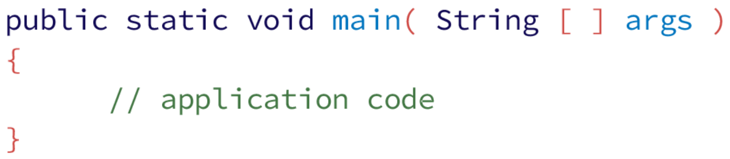
```
';' expected
```

Description

In the method body, a method can declare variables, call other methods, and use any of the program structures we've discussed: *if/else* statements, *while* loops, *for* loops, *switch* statements, and *do/while* loops.

All objects of a class share one copy of the class methods.

We have actually written methods already. For example, we've written the method *main*. Its definition looks like this:

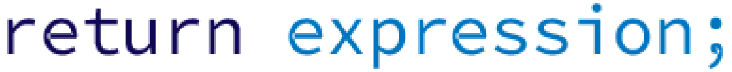


```
public static void main( String [ ] args )
{
      // application code
}
```

Description

We know that the *static* keyword means that the Java Virtual Machine (JVM) can call *main* to start the application running without first instantiating an object. The return type is *void* because *main* does not return a value. The parameter list expects one argument, a *String* array.

We have not previously written a value-returning method. A value-returning method sends back its results to the caller using a *return* statement in the method body. The syntax for the *return* statement is

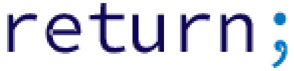


```
return expression;
```

Description

As you would expect, the data type of the expression must match the return type of the method. Recall that a value-returning method is called from an expression, and when the method completes its operation, its return value replaces the method call in the expression.

If the data type of the method is *void*, as in *main*, we have a choice of using the *return* statement without an expression, as in this statement:



```
return;
```

Description

or omitting the *return* statement altogether. Given that control automatically returns to the caller when the end of the method is reached, most programmers omit the *return* statement in *void* methods. Optionally, a *return* statement can be used if needed to exit a method before reaching the end of the method's code.

```
Instantiate sedan
Instantiate suv
Instantiate mini
```

7.4 Writing Constructors

A constructor is a special method that is called when an object is instantiated using the *new* keyword. A class can have several constructors. The job of the class constructors is to initialize the fields of the new object.

The syntax for a constructor follows:

```
accessModifier ClassName( parameter list )
{
    // constructor body
}
```

Description

Notice that a constructor has the same name as the class and has no return type—not even *void*.

It's import ant to use the *public* access modifier for the constructors so that applications can instantiate objects of the class.

SOFTWARE ENGINEERING TIP

Define constructors to be *public* so that clients can instantiate objects of the class.

The constructor can either assign default values to the instance variables or the constructor can accept initial values from the client through parameters.

Providing a constructor for a class is optional. If you don't write a constructor, the compiler provides a **default constructor**, which is a constructor that takes no arguments. This default constructor assigns default initial values to all instance variables; this is called **autoinitialization**. Numeric variables are given the value of 0, characters are given the Unicode null character, *boolean* variables are given the value *false*, and object references are given the value *null*. **Table 7.2** shows the values the default constructor assigns to instance variables.

Default Initial Values of Instance Variables

Data Type	**Initial Value**
byte	0
short	0
int	0
long	0
float	0.0
double	0.0
char	null character ('\u0000')
boolean	*false*
object reference	*null*

Description

If we do provide a constructor, any instance variables our constructor does not initialize will still be given the predefined default value. Also, if we do provide a constructor, the compiler no longer generates a default constructor for us.

Example 7.1 shows Version 1 of our *Auto* class with two constructors.

Example 7.1 The *Auto* Class, Version 1

```
 1 /* Auto class, Version 1
 2    Anderson, Franceschi
 3 */
 4
 5 public class Auto
 6 {
 7    // instance variables
 8    private String model;          //  model of auto
 9    private int milesDriven;       //  number of miles driven
10    private double gallonsOfGas;   //  number of gallons of gas
11
12    // Default constructor:
13    //  initializes model to "unknown";
14    //  milesDriven is autoinitialized to 0
15    //        and gallonsOfGas to 0.0
16    public Auto( )
17    {
18       model = "unknown";
19    }
20
21    // Overloaded constructor:
22    // allows client to set beginning values for
23    //   model, milesDriven, and gallonsOfGas.
24    public Auto( String startModel,
25                 int startMilesDriven,
26                 double startGallonsOfGas )
27    {
28       model = startModel;
29
30       // validate startMilesDriven parameter
31       if ( startMilesDriven >= 0 )
32           milesDriven = startMilesDriven;
33
34       // validate startGallonsOfGas parameter
35       if ( startGallonsOfGas >= 0.0 )
36           gallonsOfGas = startGallonsOfGas;
37    }
38 }
```

Description

Our default constructor (lines 12–19) does not set values for the *milesDriven* and *gallonsOfGas* instance variables. Because *ints* and *doubles* are autoinitialized to 0 and 0.0, respectively, we just accept those default values.

However, it is necessary for our constructor to set the *model* instance variable to a valid *String* value. Because *Strings* are object references, they are autoinitialized to *null*. Any attempt to call a method using the *model* instance variable with a *null* value would generate a *NullPointerException*.

As mentioned earlier, we can provide multiple constructors for a class. We provide a second constructor (lines 21–37) that lets the client set initial values for all the instance variables. Because the class is the caretaker of its fields, it is the class's responsibility to ensure that the data for each object is valid. Thus, when the constructor sets initial values for the instance variables, it should first check whether its parameters are, indeed, valid values. What constitutes a valid value for any instance variable depends in part on the data type of the variable and in part on the class and is a design decision. For our *Auto* class, we have decided that *milesDriven* and *gallonsOfGas* cannot be negative. If the constructor finds that the *startMilesDriven* or *startGallonsOfGas* parameters are negative, it lets the instance variables be assigned default values of 0 and 0.0, respectively. Some methods in the Java Class Library generate an exception when a parameter value is invalid; others substitute a default value for the invalid parameter. Again, how your classes handle invalid argument values is a design decision.

When we provide multiple constructors, we are **overloading** a method. To overload a method, we provide a method with the same name but with a different number of parameters, or with the same number of parameters but with at least one parameter having a different data type. The name of the method, along with the number, data types, and order of its parameters, is called the method's **signature**. Thus, to overload a method, the new method must have a different signature. Notice that the return type is not part of the signature.

When a client calls a method that is overloaded, Java determines which version of the method to execute by looking at the number, data types, and order of the arguments in the method call. **Example 7.2** shows a client program that instantiates three *Auto* objects.

Example 7.2 The *Auto* Client, Version 1

```
 1 /* Auto Client, Version 1
 2    Anderson, Franceschi
 3 */
 4
 5 public class AutoClient
 6 {
 7   public static void main( String [ ] args )
 8   {
 9      System.out.println( "Instantiate sedan" );
10      Auto sedan = new Auto( );
11
12      System.out.println( "\nInstantiate suv" );
13      Auto suv = new Auto( "Trailblazer", 7000, 437.5 );
14
15      System.out.println( "\nInstantiate mini" );
16      // attempt to set invalid value for gallons of gas
17      Auto mini = new Auto( "Mini Cooper", 200, -1.0 );
18   }
19 }
```

Description

SOFTWARE ENGINEERING TIP

Provide, at the minimum, a default constructor and a constructor that accepts initial values for all instance variables.

Line 10 causes the default constructor to be called because no arguments are passed to the constructor. Line 13 causes the overloaded constructor to be called because it passes three arguments to the constructor. If the client attempted to instantiate a new object with a number of parameters other than 0 or 3, the compiler would generate an error because there is no constructor that matches those arguments. In general, the arguments sent to an overloaded method must match the formal parameters of some version of that method.

The number of constructors we provide is a design decision and depends on the class. Providing multiple constructors gives the client a choice of ways to create an object. It is good practice to provide, at minimum, a default constructor. The reason for this will become clear as we explore classes in more depth. It is also good practice to provide another constructor that accepts values for all the instance variables.

On line 17, we instantiate an *Auto* object with an invalid argument for gallons of gas. The object is still created, but the value of its *gallonsOfGas* instance variable is autoinitialized to 0.0. The output of Example 7.2 is shown in **Figure 7.1**.

Figure 7.1 Output from *Auto* Client, Version 1

Description

Beware of this common error: declaring a *void* return type for a constructor. Remember that constructors have no return type at all. For example, the following invalid constructor definition declares a return type of *void*:

```
// Error! void return value specified
public void Auto( String model,
                  int startMilesDriven,
                  double startGallonsOfGas )
{
     // body of constructor
}
```

Description

This is a difficult error to find. The class file will compile without an error because the compiler doesn’t recognize this method as a constructor. Instead, the client program will get a compiler error when it attempts to instantiate an *Auto* object. For example, this statement in a client program

```
Auto gm = new Auto( "Prius", 350, 15.5 );
```

Description

would generate this compiler error:

```
AutoClient.java:13: error: constructor Auto in class Auto cannot be applied to given types;
      Auto gm = new Auto( "Prius", 350, 15.5 );
                ^
  required: no arguments
  found: String,int,double
  reason: actual and formal argument lists differ in length
1 error
```

Description

Notice that both constructors access the instance variables directly. Remember that instance variables have class scope, which means that they can be accessed anywhere in the class. Thus, any method of the class can access any of the instance variables directly. In our *Auto* class, any method can access the instance variables *model, milesDriven*, and *gallonsOfGas*.

COMMON ERROR TRAP

Specifying a return value for a constructor will cause a compiler error in the client program when the client attempts to instantiate an object of that class.

Methods have class scope as well. Any method can call any of the methods in the class, regardless of whether the methods have been declared *private, public*, or *protected*.

In addition to accessing the instance variables, a method can also access its own parameters. When a method begins executing, its parameters have been declared and have been given the values of the arguments sent by the caller of the method.

The parameters have **local scope** in that a method can access its parameters directly. We call this local scope because the parameters can be accessed only in that method; that is, a method can access its own parameters, but attempting to access another method's parameters generates a compiler error.

Similarly, the method can define variables to be used within the method. These variables also have local scope and are accessible from the point of definition until the end of the method or the end of the block in which the variable was defined, whichever comes first.

Table 7.3 summarizes the rules of scope.

Rules of Scope

A method in a class can access

- the instance variables of its class
- any parameters sent to the method
- any variable the method declares within its body from the point of declaration until the end of the method or until the end of the block in which the variable was declared, whichever comes first

- any methods in the class

Description

Attempting to use an identifier that is not in scope will generate the following compiler error:

```
cannot find symbol
```

Description

When the client in Example 7.2 runs, it instantiates three objects, but there is nothing more our application can do with them. To allow our client to manipulate the *Auto* objects further, we need to provide more methods.

with these end-of-chapter questions

7.19.1	Multiple Choice Exercises Questions 1, 2, 3, 4, 5, 6, 7
7.19.3	Fill In the Code Questions 28, 30, 31
7.19.5	Debugging Area Questions 47, 48, 49
7.19.8	Technical Writing Question 77

```
sedan: model is unknown
 miles driven is 0
 gallons of gas is 0.0
suv: model is Trailblazer
 miles driven is 7000
 gallons of gas is 437.5
```

7.5 Writing Accessor Methods

Because clients cannot directly access *private* instance variables of a class, classes usually provide *public* accessor methods for the instance variables. These methods have a simple, almost trivial, standard form:

```
public returnType getInstanceVariable( )
{
   return instanceVariable;
}
```

The standard name of the method is *get*, followed by the instance variable's name with an initial capital letter. Because the method names usually start with "get," accessor methods are often called **getters**. The method takes no arguments and simply returns the current value of the instance variable. Thus, the return type is the same data type as the instance variable.

You can see this simple pattern in the accessor methods for Version 2 of our *Auto* class, shown in **Example 7.3** (lines 39–58).

Example 7.3 *Auto* Class, Version 2

```java
/* Auto class, Version 2
   Anderson, Franceschi
*/

public class Auto
{
    // instance variables
    private String model;          //  model of auto
    private int milesDriven;       //  number of miles driven
    private double gallonsOfGas;   //  number of gallons of gas

    // Default constructor:
    //  initializes model to "unknown";
    //  milesDriven is autoinitialized to 0
    //      and gallonsOfGas to 0.0
    public Auto( )
    {
       model = "unknown";
    }

    // Overloaded constructor:
    // allows client to set beginning values for
    //   model, milesDriven, and gallonsOfGas.
    public Auto( String startModel,
                 int startMilesDriven,
                 double startGallonsOfGas )
    {
       model = startModel;

       // validate startMilesDriven parameter
       if ( startMilesDriven >= 0 )
          milesDriven = startMilesDriven;

       // validate startGallonsOfGas parameter
       if ( startGallonsOfGas >= 0.0 )
          gallonsOfGas = startGallonsOfGas;
    }

    // Accessor method:
    // returns current value of model
    public String getModel( )
    {
       return model;
    }

    // Accessor method:
    // returns current value of milesDriven
    public int getMilesDriven( )
    {
       return milesDriven;
    }

    // Accessor method:
    //  returns current value of gallonsOfGas
    public double getGallonsOfGas( )
    {
       return gallonsOfGas;
    }
}
```

Description

In the client code in **Example 7.4**, we’ve added a few statements to call the accessor methods for the two *Auto* objects we’ve instantiated. Then we print the values, as shown in **Figure 7.2**.

Example 7.4 *Auto* Client, Version 2

```java
/* Auto Client, Version 2
   Anderson, Franceschi
*/

public class AutoClient
{
    public static void main( String [ ] args )
    {
        Auto sedan = new Auto( );
       String sedanModel = sedan.getModel( );
       int sedanMiles = sedan.getMilesDriven( );
       double sedanGallons = sedan.getGallonsOfGas( );
       System.out.println( "sedan: model is " + sedanModel
                  + "\n miles driven is " + sedanMiles
                  + "\n gallons of gas is " + sedanGallons );

       Auto suv = new Auto( "Trailblazer", 7000, 437.5 );
       String suvModel = suv.getModel( );
       int suvMiles = suv.getMilesDriven( );
       double suvGallons = suv.getGallonsOfGas( );
       System.out.println( "suv: model is " + suvModel
                  + "\n miles driven is " + suvMiles
                  + "\n gallons of gas is " + suvGallons );
   }
}
```

Description

Figure 7.2 Output from *Auto* Client, Version 2

Description

Because the *sedan* object was instantiated by calling the default constructor, its model is *unknown* and the miles driven and gallons of gas are set to default values. On the other hand, the *suv* object data reflects the values sent to the overloaded constructor when the *suv* object was instantiated.

SOFTWARE ENGINEERING TIP

Provide *public* accessor methods for any instance variable for which the client should be able to retrieve the value. Each accessor method returns the current value of the corresponding instance variable.

Thus, Version 2 of our *Auto* class lets our clients instantiate objects and get the values of the instance variables. But we still need to give the client a way to change the instance variables. In order to do this, we provide mutator methods.

```
suv: model is Trailblazer
 miles driven is 7000
 gallons of gas is 437.5

suv: model is Sportage
 miles driven is 200
 gallons of gas is 10.5

suv: model is Sportage
 miles driven is 200
 gallons of gas is 10.5
```

7.6 Writing Mutator Methods

As we have discussed, we declare the instance variables as *private* to encapsulate the data of the class. We allow only the class methods to directly set the values of the instance variables. Thus, it is customary to provide a *public* **mutator** method for any instance variable that the client will be able to change. Because the method names usually start with “set,” mutator methods are often called **setters**.

The general form of a mutator method is the following:

```
public void setInstanceVariable( dataType newValue )
{
    // validate newValue, then assign to the instance variable
}
```

Description

SOFTWARE ENGINEERING TIP

Provide a mutator method for any instance variable that you want to allow the client to change.

We declare mutator methods as *public* so that client programs can use the methods to change the values of the instance variables. We do not return a value, so we declare the return type as *void*. By convention, the name of each mutator method starts with the lowercase word *set* followed by the instance variable name with an initial capital letter. For obvious reasons, the data type of the method’s parameter should match the data type of the instance variable being set.

Whenever possible, the body of your mutator method should validate the parameter value passed by the client. If the parameter value is valid, the mutator assigns that value to the instance variable.

Example 7.5 shows Version 3 of our *Auto* class.

Example 7.5 *Auto* Class, Version 3

```java
/* Auto class, Version 3
   Anderson, Franceschi
*/

public class Auto
{
    // instance variables
    private String model;          //  model of auto
    private int milesDriven;       //  number of miles driven
    private double gallonsOfGas;   //  number of gallons of gas

    // Default constructor:
    //  initializes model to "unknown";
    //  milesDriven is autoinitialized to 0
    //       and gallonsOfGas to 0.0
    public Auto( )
    {
       model = "unknown";
    }

    // Overloaded constructor:
    // allows client to set beginning values for
    //   model, milesDriven, and gallonsOfGas.
    public Auto( String startModel,
                 int startMilesDriven,
                 double startGallonsOfGas )
    {
       model = startModel;
       setMilesDriven( startMilesDriven );
       setGallonsOfGas( startGallonsOfGas );
    }

    // Accessor method:
    // returns current value of model
    public String getModel( )
    {
      return model;
    }

    // Accessor method:
    // returns current value of milesDriven
    public int getMilesDriven( )
    {
      return milesDriven;
    }

    // Accessor method:
    //  returns current value of gallonsOfGas
    public double getGallonsOfGas( )
    {
      return gallonsOfGas;
    }

    // Mutator method:
    // allows client to set model
    public void setModel( String newModel )
    {
      model = newModel;
    }

    // Mutator method:
    // allows client to set value of milesDriven;
    // if new value is not less than 0
    public void setMilesDriven( int newMilesDriven )
    {
      if ( newMilesDriven >= 0 )
       milesDriven = newMilesDriven;
    }

    // Mutator method:
    // allows client to set value of gallonsOfGas;
    // if new value is not less than 0.0
    public void setGallonsOfGas( double newGallonsOfGas )
    {
      if ( newGallonsOfGas >= 0.0 )
        gallonsOfGas = newGallonsOfGas;
    }
}
```

Description

The mutator methods for the *milesDriven* (lines 61–68) and *gallonsOfGas* (lines 70–77) instance variables validate that the parameter value is greater than or equal to 0. If the parameter value is less than 0, the methods do not change the value of the instance variable. In previous versions of our *Auto* class, the constructor performed the same validation. Now that the mutator methods perform this validation, the constructor can call the mutator methods (lines 29–30) passing its parameters through to the mutator methods. In this way, we eliminate duplicate code; the validation of each parameter's value is performed in one place. If later we decide to impose other restrictions on any instance variable's value, we will need to change the code in only one place. In this way, a client cannot set invalid values for *milesDriven* or *gallonsOfGas*, either when the object is instantiated or by calling a mutator method.

In **Example 7.6**, our client instantiates one *Auto* object, *suv* (line 9), and prints the values of its instance variables (lines 11–14). Then we call each mutator method, setting new values for each instance variable (lines 16–19). We again print the values of the instance variables (lines 21–24) to show that the values have been changed. Then, in line 27, we attempt to set an invalid value for *milesDriven*. As **Figure 7.3** shows, the mutator method does not change the value, which we verify by again printing the values of the instance variables (lines 28–31).

Example 7.6 *Auto* Client, Version 3

```
 1 /* Auto Client, Version 3
 2    Anderson, Franceschi
 3 */
 4
 5 public class AutoClient
 6 {
 7   public static void main( String [ ] args )
 8   {
 9      Auto suv = new Auto( "Trailblazer", 7000, 437.5 );
10
11      // print initial values of instance variables
12      System.out.println( "suv: model is " + suv.getModel( )
13             + "\n miles driven is " + suv.getMilesDriven( )
14             + "\n gallons of gas is " + suv.getGallonsOfGas( ) );
15
16      // call mutator method for each instance variable
17      suv.setModel( "Sportage" );
18      suv.setMilesDriven( 200 );
19      suv.setGallonsOfGas( 10.5 );
20
21      // print new values of instance variables
22      System.out.println( "\nsuv: model is " + suv.getModel( )
23             + "\n miles driven is " + suv.getMilesDriven( )
24             + "\n gallons of gas is " + suv.getGallonsOfGas( ) );
25
26      // attempt to set invalid value for milesDriven
27      suv.setMilesDriven( -1 );
28      // print current values of instance variables
29      System.out.println( "\nsuv: model is " + suv.getModel( )
30             + "\n miles driven is " + suv.getMilesDriven( )
31             + "\n gallons of gas is " + suv.getGallonsOfGas( ) );
32   }
33 }
```

Description

Figure 7.3 Output from *Auto* Client, Version 3

Description

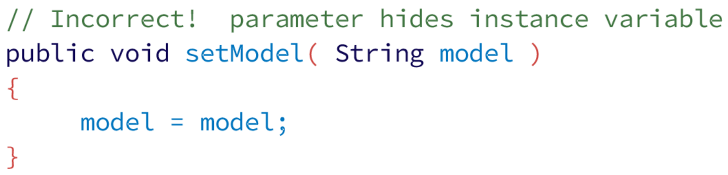
SOFTWARE ENGINEERING TIP

Write the validation code for instance variables in mutator methods and have the constructor call the mutator methods to set initial values.

When a method begins executing, the parameters have been defined and have been assigned the values sent by the client. When the client calls the *setModel* method at line 17, the *newModel* parameter has the value *Sportage* when the method starts executing.

A common error in writing mutator methods is using the instance variable name for the parameter name. When a method parameter has the same name as an instance variable, the parameter hides the instance variable. In other words, the parameter has **name precedence**, so any reference to that name refers to the parameter, not to the instance variable.

For example, the intention in this incorrectly coded method is to set a new value for the *model* instance variable:

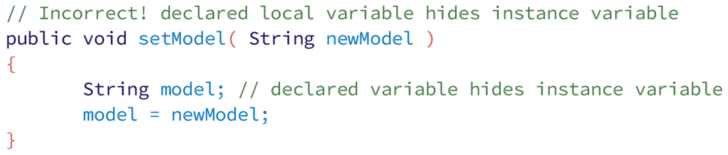


```
// Incorrect!  parameter hides instance variable
public void setModel( String model )
{
      model = model;
}
```

Description

Because the parameter, *model*, has the same identifier as the *model* instance variable, the result of this method is to assign the value of the parameter to the parameter! This is called a ***No-op***, which stands for “No operation,” because the statement has no effect. To avoid this logic error, we can choose a different name for the parameter. To avoid name conflicts, we name each parameter using the pattern *newInstanceVariable*.

A similar common error is to declare a local variable with the same name as the instance variable, as shown in the following incorrectly coded method:



```
// Incorrect! declared local variable hides instance variable
public void setModel( String newModel )
{
       String model; // declared variable hides instance variable
       model = newModel;
}
```

Description

COMMON ERROR TRAP

Be aware that a method parameter or local variable that has the same name as an instance variable hides the instance variable.

Any variable that a method declares is a local variable because its scope is local to the method. Thus, the declared variable, *model*, is a local variable to the *setModel* method.

With the preceding code, the *model* local variable hides the instance variable with the same name, so the method assigns the parameter value to the local variable, not to the instance variable. The result is that the value of the *model* instance variable is unchanged.

COMMON ERROR TRAP

Do not declare the parameters of a method inside the method body. When the method begins executing, the parameters exist and have been assigned the values set by the client in the method call.

The instance variable, *model*, is defined already in the class. Thus, the method should simply assign the parameter value to the instance variable without attempting to declare the instance variable (again) in the method.

Finally, another common error is declaring the parameter, as shown below:

```
// Incorrect! Declaring the parameter; parameters are declared already
public void setModel( String newModel )
{
       String newModel; // local variable has same name as parameter
       model = newModel;
}
```

Description

This code generates this compiler error:

```
variable newModel is already defined in method setModel(String)
```

Description

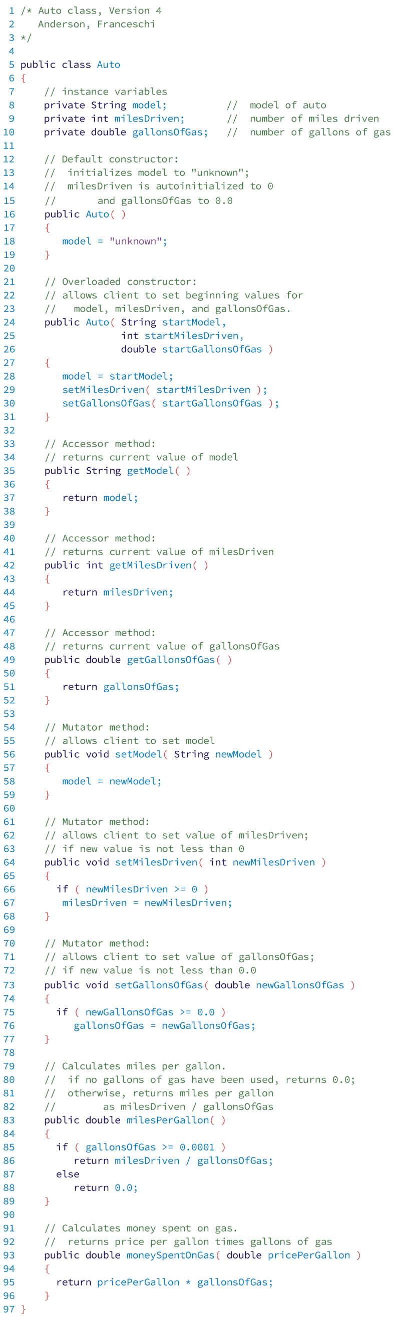
```
Mileage for suv is 16.0
Gas cost for suv is $1,220.62
```

7.7 Writing Data Manipulation Methods

Now we finally get down to the business of the class. Usually you will define a class not only to encapsulate the data, but also to provide some service. Thus, you would provide one or more methods that perform the functionality of the class. These methods might calculate a value based on the instance variables and/or parameters, or manipulate the instance variables in some way. The API of these methods depends on the function being performed. If a method merely manipulates the instance variables, it requires no parameters because instance variables are accessible from any method and, therefore, are in scope.

For example, in our *Auto* class, part of the functionality of our class is to calculate miles per gallon and the gas cost, so we provide the *milesPerGallon* and *moneySpentOnGas* methods in our *Auto* class, Version 4, shown in Example 7.7.

Example 7.7 *Auto* Class, Version 4



```java
/* Auto class, Version 4
   Anderson, Franceschi
*/

public class Auto
{
    // instance variables
    private String model;          //  model of auto
    private int milesDriven;       //  number of miles driven
    private double gallonsOfGas;   //  number of gallons of gas

    // Default constructor:
    //  initializes model to "unknown";
    //  milesDriven is autoinitialized to 0
    //       and gallonsOfGas to 0.0
    public Auto( )
    {
       model = "unknown";
    }

    // Overloaded constructor:
    // allows client to set beginning values for
    //   model, milesDriven, and gallonsOfGas.
    public Auto( String startModel,
                 int startMilesDriven,
                 double startGallonsOfGas )
    {
       model = startModel;
       setMilesDriven( startMilesDriven );
       setGallonsOfGas( startGallonsOfGas );
    }

    // Accessor method:
    // returns current value of model
    public String getModel( )
    {
       return model;
    }

    // Accessor method:
    // returns current value of milesDriven
    public int getMilesDriven( )
    {
       return milesDriven;
    }

    // Accessor method:
    // returns current value of gallonsOfGas
    public double getGallonsOfGas( )
    {
       return gallonsOfGas;
    }

    // Mutator method:
    // allows client to set model
    public void setModel( String newModel )
    {
       model = newModel;
    }

    // Mutator method:
    // allows client to set value of milesDriven;
    // if new value is not less than 0
    public void setMilesDriven( int newMilesDriven )
    {
      if ( newMilesDriven >= 0 )
       milesDriven = newMilesDriven;
    }

    // Mutator method:
    // allows client to set value of gallonsOfGas;
    // if new value is not less than 0.0
    public void setGallonsOfGas( double newGallonsOfGas )
    {
      if ( newGallonsOfGas >= 0.0 )
         gallonsOfGas = newGallonsOfGas;
    }

    // Calculates miles per gallon.
    //  if no gallons of gas have been used, returns 0.0;
    //  otherwise, returns miles per gallon
    //        as milesDriven / gallonsOfGas
    public double milesPerGallon( )
    {
      if ( gallonsOfGas >= 0.0001 )
         return milesDriven / gallonsOfGas;
      else
         return 0.0;
    }

    // Calculates money spent on gas.
    //  returns price per gallon times gallons of gas
    public double moneySpentOnGas( double pricePerGallon )
    {
      return pricePerGallon * gallonsOfGas;
    }
}
```

Description

Our class now provides the method to calculate mileage and estimate gas cost for an *Auto* object. The *milesPerGallon* method (lines 79–89) needs no parameters since it accesses only instance variables of the class, which are in scope. As you can see from the code, we guard against dividing by 0 by checking the value of *gallonsOfGas* before using it as the divisor. If *gallonsOfGas* is not equal to zero, we divide *milesDriven* by *gallonsOfGas* and return the result as a *double*. Otherwise, we return 0.0.

The *moneySpentOnGas* method (lines 91–96) accepts one parameter that represents the average price of a gallon of gas. It returns the amount of money spent on gas, calculated by multiplying that parameter by the *gallonsOfGas* instance variable.

Notice that we do not format the returned values from *milesPerGallon* or *moneySpentOnGas*. Returning a *double* allows the client to use those values in further calculations. If the client wants to display the values to the user, the client can format the values at that time.

Example 7.8 shows a client program that instantiates an *Auto* object, calls the *milesPerGallon* and *moneySpentOnGas* methods, and prints their return value, as shown in **Figure 7.4**.

Example 7.8 *Auto* Client, Version 4

```
/* Auto Client, Version 4
   Anderson, Franceschi
*/

import java.text.DecimalFormat;
import java.text.NumberFormat;

public class AutoClient
{
   public static void main( String [ ] args )
   {
     Auto suv = new Auto( "Trailblazer", 7000, 437.5 );

     double mileage = suv.milesPerGallon( );
     DecimalFormat mpgFormat = new DecimalFormat( "0.0" );
     System.out.println( "Mileage for suv is "
                          + mpgFormat.format( mileage ) );

     double gasCost = suv.moneySpentOnGas( 2.79 );
     NumberFormat money = NumberFormat.getCurrencyInstance( );
     System.out.println( "Gas cost for suv is "
                          + money.format( gasCost ) );
   }
}
```

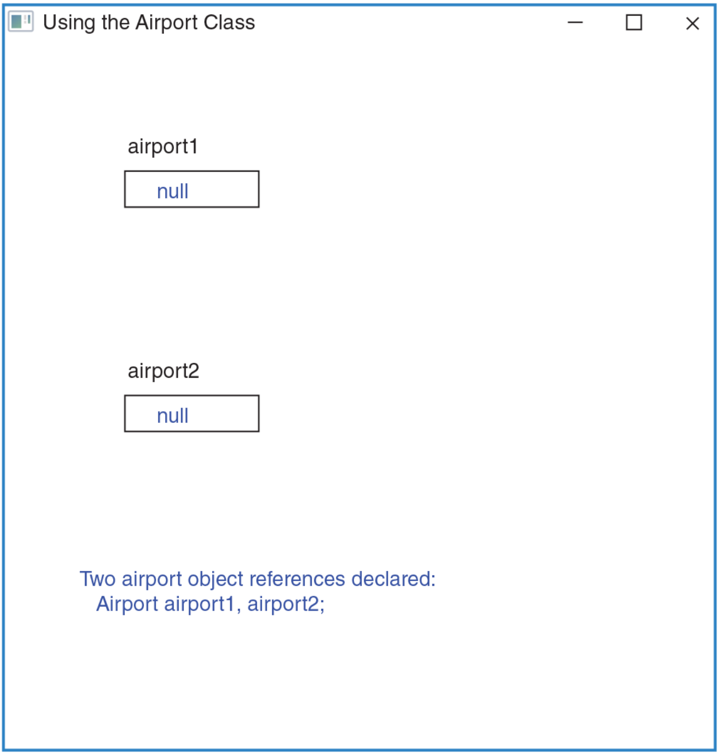
Description

Figure 7.4 Output from *Auto* Client, Version 4

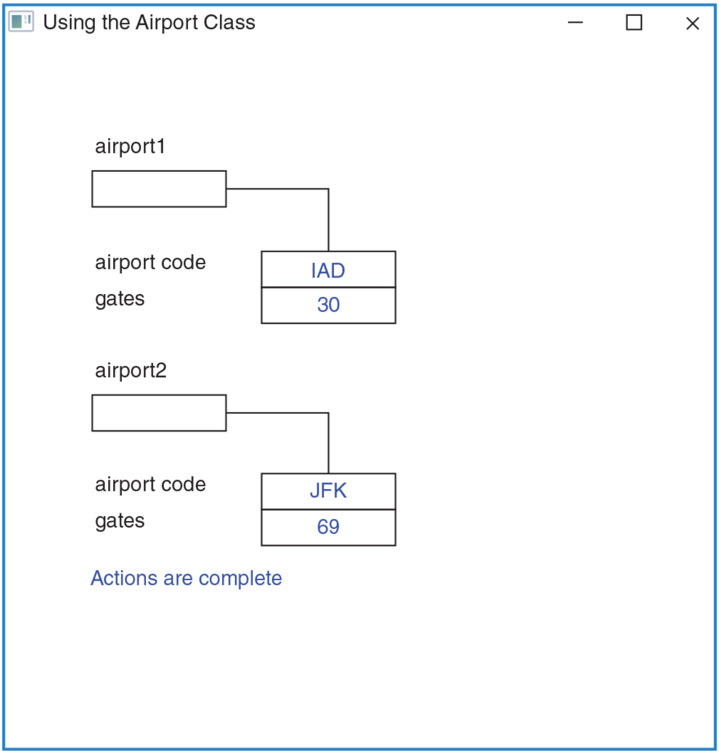
Description

with these end-of-chapter questions

7.19.1	Multiple Choice Exercises Questions 8, 9, 10, 11, 12, 13
7.19.2	Reading and Understanding Code Questions 17, 18, 19, 20, 24, 26
7.19.3	Fill In the Code Questions 32, 33, 36, 37
7.19.4	Identifying Errors in Code Questions 38, 39, 43, 45
7.19.5	Debugging Area Question 52


Using the Airport Class
airport1
null
airport2
null

Two airport object references declared:
Airport airport1, airport2;



Actions are complete

7.8 Programming Activity 1: Writing a Class Definition, Part 1

In this programming activity, you will write the methods for an *Airport* class. Then you will run a prewritten client program that instantiates several *Airport* objects, calls the methods that you have written, and displays the values of the objects' data.

The *Airport* class has two instance variables: the airport code and the number of gates.

In this chapter's Programming Activity 1 folder in the supplied code files, you will find multiple files. Copy these files to a folder on your computer. Note that all files should be in the same folder.

Load the *Airport.java* source file; you'll notice that the class already contains some source code. The method names and APIs are described in comments. Your job is to define the instance variables and write the methods. It is important that you define the method headers exactly as described, including method name, return value, and parameters, because our *AirportDrawing* class will call each method to test it. Search for five asterisks in a row (*****). This will position you at the seven places in the class definition where you will add your code. The *Airport.java* code is shown here in **Example 7.9**.

Example 7.9 *Airport.java*

```java
/* Airport class
   Anderson, Franceschi
*/

public class Airport
{
  // 1. ***** Define the instance variables  *****
  //  airportCode is a String
  //  gates is an integer



   // 2. ***** Write this method *****
   // Default constructor:
   // method name: Airport
   // return value:  none
   // parameters: none
   // function: sets the airportCode to an empty String



   // 3. ***** Write this method *****
   // Overloaded constructor:
   // method name: Airport
   // return value: none
   // parameters:  a String startAirportCode and an int startGates
   // function:
   //      calls the setAirportCode method,
   //      passing startAirportCode parameter;
   //      calls the setGates method, passing startGates parameter




   // 4. ***** Write this method *****
   // Accessor method for the airportCode instance variable
   // method name: getAirportCode
   // return value: String
   // parameters: none
   // function: returns airportCode



   // 5. ***** Write this method *****
   // Accessor method for the gates instance variable
   // method name: getGates
   // return value: int
   // parameters: none
   // function: returns gates



   // 6. ***** Write this method *****
   // Mutator method for the airportCode instance variable
   // method name: setAirportCode
   // return value: void
   // parameters: String newAirportCode
   // function: assigns airportCode the value of the
   //       newAirportCode parameter



   // 7. ***** Write this method *****
   // Mutator method for the gates instance variable
   // method name: setGates
   // return value:  void
   // parameters: int newGates
   // function: validates the newGates parameter.
   //   if newGates is greater than or equal to 0,
   //       sets gates to newGates;
   //       otherwise, does not change the value of gates



  }  // end of Airport class definition
```

Description

When you finish writing the methods for the *Airport* class, compile the source file. When *Airport.java* compiles without errors, load the *AirportPractice1Application.java* file. This source file contains *main*, so you will execute the application from this file. When the application begins, you should see the window shown in **Figure 7.5**.

Figure 7.5 Programming Activity 1 Opening Window

Description

As you can see, the *AirportDrawing* has declared two *Airport* object references, *airport1* and *airport2*. The references are *null* because no *Airport* objects have been instantiated.

The client application will instantiate the *Airport* objects and call the methods you have written for the *Airport* class. As the application does its work, it displays a status message at the bottom of the window that indicates which method it has called. It also displays the current values of both *Airport* objects. You can check your work by comparing the values in the objects with the status message. **Figure 7.6** shows the *AirportPractice1Application* application when it has finished instantiating *Airport* objects and calling *Airport* methods.

Figure 7.6 *AirportPractice1Application* When Complete

Description

DISCUSSION QUESTIONS

1. Why does the default constructor need to set a value for the airport code?
2. Explain the importance of using standard naming conventions for accessor and mutator methods.

```
car is Auto@4a574795
this is Auto@4a574795
Miles driven is 67
Gallons of gas is 3.4
```

7.9 The Object Reference *this*

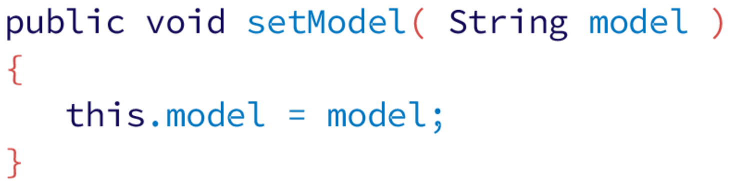
When an object is instantiated, a copy of each of the instance variables is created. However, all objects of a class share one copy of the methods. How, then, does a method know for which object the method was called? In other words, how does a method know which object's data it should get, set, or use to calculate a value? The answer is the special object reference named ***this***.

When a method begins executing, the JVM sets the object reference, *this*, to refer to the object for which the method has been called. That object reference is called the **implicit parameter**. When a method references an instance variable, it will access the instance variable that belongs to the object that the implicit parameter references. In other words, by default, any instance variable referred to by a method is considered to be *this.instanceVariable.*

Preceding an instance variable name with *this* is optional; when just the instance variable name is used (without any object reference), *this* is assumed. Consequently, we usually omit the *this* reference and use just the instance variable name.

However, in methods where you need to avoid ambiguity in variable names, you can precede an instance variable name with *this.* That approach comes in handy as a way to avoid one of the common errors we discussed earlier in the chapter: A parameter of a mutator method with the same name as the instance variable hides the instance variable. We can eliminate this problem by using the *this* reference with the instance variable, which effectively uncovers the instance variable name.

For example, some programmers would code the *setModel* mutator as follows:

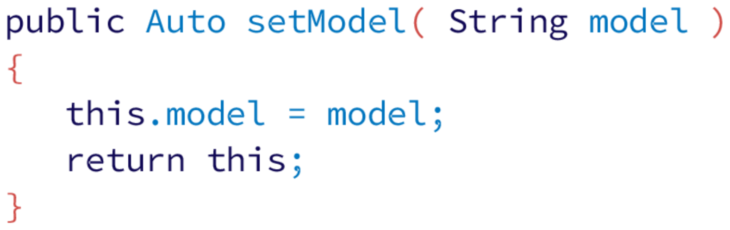


```
public void setModel( String model )
{
    this.model = model;
}
```

Description

Here we give the parameter, *model*, the same name as the instance variable it represents. Then in the assignment statement, we use the *this* reference to distinguish the instance variable from the parameter. Now it is clear that the parameter value, *model,* should be assigned to the instance variable, *this.model.*

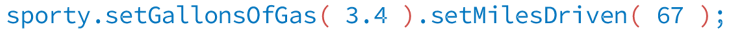
Furthermore, some programmers, particularly in the Android community, would code the *setModel* mutator so that it returns a reference to *this Auto*, as follows:



```
public Auto setModel( String model )
{
    this.model = model;
    return this;
}
```

Description

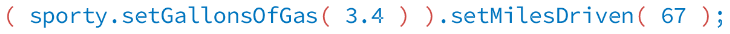
The benefit of returning *this* is that we can chain method calls. For example, assuming the *setMilesDriven* and *setGallonsOfGas* mutators also return *this* and *sporty* is an *Auto* object, we could chain method calls as follows:



```
sporty.setGallonsOfGas( 3.4 ).setMilesDriven( 67 );
```

Description

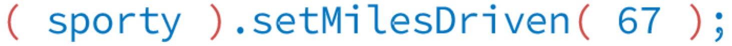
Since the . (dot) operator associates left to right, the above statement is equivalent to:



```
( sporty.setGallonsOfGas( 3.4 ) ).setMilesDriven( 67 );
```

Description

Since the *setGallonsOfGas* method returns *this*—that is, the object reference that called it—it returns *sporty* in the above statement. Thus, the expression `sporty.setGallonsOfGas ( 3.4 )` evaluates to *sporty*, which in turns calls the method *setMilesDriven* with argument *67*, as shown below:



```
( sporty ).setMilesDriven( 67 );
```

Description

Example 7.10 shows Version 5 of our *Auto* class. The three mutators (lines 54–60, 62–70, and 72–81) all use and return the *this* reference. At line 77, we added an output statement inside the *setGallonsOfGas* method to help understand that *this* is a reference to the object reference that called the method.

Example 7.10 *Auto* Class, Version 5

```
/* Auto class, Version 5
   Anderson, Franceschi
*/

public class Auto
{
    // instance variables
    private String model;          //  model of auto
    private int milesDriven;       //  number of miles driven
    private double gallonsOfGas;   //  number of gallons of gas

    // Default constructor:
    //  initializes model to "unknown";
    //  milesDriven is autoinitialized to 0
    //        and gallonsOfGas to 0.0
    public Auto( )
    {
       model = "unknown";
    }

    // Overloaded constructor:
    // allows client to set beginning values for
    //   model, milesDriven, and gallonsOfGas.
    public Auto( String startModel,
                 int startMilesDriven,
                 double startGallonsOfGas )
    {
       model = startModel;
       setMilesDriven( startMilesDriven );
       setGallonsOfGas( startGallonsOfGas );
    }

    // Accessor method:
    // returns current value of model
    public String getModel( )
    {
       return model;
    }

    // Accessor method:
    // returns current value of milesDriven
    public int getMilesDriven( )
    {
       return milesDriven;
    }

    // Accessor method:
    // returns current value of gallonsOfGas
    public double getGallonsOfGas( )
    {
       return gallonsOfGas;
    }

    // Mutator method:
    // allows client to set model
    public Auto setModel( String model )
    {
       this.model = model;
       return this;
    }

    // Mutator method:
    // allows client to set value of milesDriven;
    // if new value is not less than 0
    public Auto setMilesDriven( int milesDriven )
    {
      if ( milesDriven >= 0 )
         this.milesDriven = milesDriven;
      return this;
    }

    // Mutator method:
    // allows client to set value of gallonsOfGas;
    // if new value is not less than 0.0
    public Auto setGallonsOfGas( double gallonsOfGas )
    {
      System.out.println( "this is " + this );
      if ( gallonsOfGas >= 0.0 )
         this.gallonsOfGas = gallonsOfGas;
      return this;
    }

    // Calculates miles per gallon.
    //  if no gallons of gas have been used, returns 0.0;
    //  otherwise, returns miles per gallon
    //        as milesDriven / gallonsOfGas
    public double milesPerGallon( )
    {
      if ( gallonsOfGas >= 0.0001 )
         return milesDriven / gallonsOfGas;
      else
         return 0.0;
    }

    // Calculates money spent on gas.
    //  returns price per gallon times gallons of gas
    public double moneySpentOnGas( double pricePerGallon )
    {
      return pricePerGallon * gallonsOfGas;
    }
 }
```

Description

At line 11 of **Example 7.11**, we demonstrate how we can chain calls to methods that return the *this* reference. At line 10, we output car. As we can see in **Figure 7.7**, the hexadecimal representation of *car* is the same as *this* inside the *setGallonsOfGas* method. This shows that *car* (in *main*) and *this* (in *setGallonsOfGas*) are the same object reference.

Example 7.11 *Auto* Client, Version 5

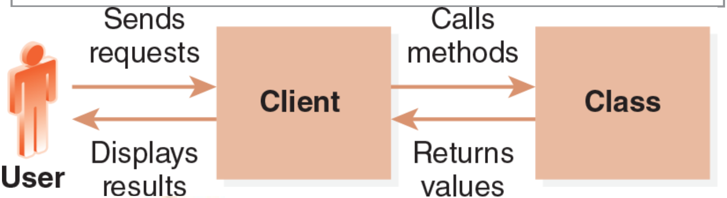
```
 1 /* Auto Client, Version 5
 2    Anderson, Franceschi
 3 */
 4
 5 public class AutoClient
 6 {
 7    public static void main( String [ ] args )
 8    {
 9      Auto car = new Auto( );
10      System.out.println( "car is " + car );
11      car.setGallonsOfGas( 3.4 ).setMilesDriven( 67 );
12
13      int carMiles = car.getMilesDriven( );
14      double carGallons = car.getGallonsOfGas( );
15
16      System.out.println( "Miles driven is " + carMiles
17              + "\nGallons of gas is " + carGallons );
18    }
19 }
```

Description

Figure 7.7 Output from *Auto* Client, Version 5

Description

```
Model: Spyder; miles driven: 0; gallons of gas: 0.00
Model: Accent; miles driven: 0; gallons of gas: 0.00
sporty and compact are not equal
```



7.10 The *toString* and *equals* Methods

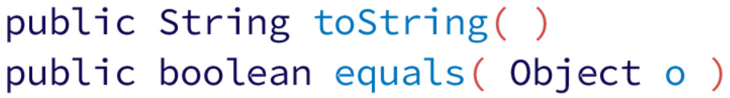
In addition to constructors, mutator methods, and accessor methods, a well-designed class usually implements the *toString* and *equals* methods.

The *toString* method is called automatically when an object reference is used as a *String*. For example, the *toString* method for an object is called when the object reference is used with, or as, a parameter to *System.out.println.* The function of the *toString* method is to return a printable representation of the object data.

The *equals* method is designed to compare two objects for equality; that is, it typically returns *true* if the corresponding instance variables in both objects are equal in value. The *equals* method takes an *Object* reference parameter that is expected to be an *Auto* reference and returns *true* if the values of its fields are equal to the values of the fields of this *Auto* object, *false* otherwise.

All classes inherit a version of the *toString* and the *equals* methods from the *Object* class, but these versions do not provide the functionality we described earlier. Thus, it is good practice to provide new versions of these methods. To do that, we use the same header as the methods in the *Object* class but provide a new method body. This is called **overriding a method.**

The APIs of the *toString* and *equals* methods are the following:



```
public String toString( )
public boolean equals( Object o )
```

Description

Example 7.12 shows the code added in Version 6 of the *Auto* class with implementations of the *toString* method (lines 103–112) and the *equals* method (lines 114–131). When overriding a method, a good programming practice is add the *@Override* annotation before the method header (lines 104 and 116). It tells the compiler that the method that follows is overriding a method inherited from another class. And in turn, the compiler warns us if our method header does not in fact override the inherited method.

Example 7.12 *Auto* Class, Version 6

```
  1 /* Auto class, Version 6
  2    Anderson, Franceschi
  3 */
  4
  5 import java.text.DecimalFormat;
  6
  7 public class Auto
  8 {
  9
       ...
103    // toString: returns a String of instance variable values
104    @Override
105    public String toString( )
106    {
107       DecimalFormat gallonsFormat = new DecimalFormat( "#0.00" );
108       return "Model: " + model
109             + "; miles driven: " + milesDriven
110             + "; gallons of gas: "
111             + gallonsFormat.format( gallonsOfGas );
112    }
113
114    // equals: returns true if fields of parameter object
115    //         are equal to fields in this object
116    @Override
117    public boolean equals( Object o )
118    {
119       if ( o instanceof Auto objAuto )
120       {
121          if ( model.equals( objAuto.model )
122               && milesDriven == objAuto.milesDriven
123               && Math.abs( gallonsOfGas - objAuto.gallonsOfGas )
124                                                 < 0.0001 )
125               return true;
126          else
127               return false;
128       }
129       else
130          return false;
131    }
132 }
```

Description

In the *toString* method (lines 103–111), we begin by instantiating a *DecimalFormat* object for formatting the gallons of gas as a floating-point number with one decimal place. Note that *gallonsFormat* is a local variable for the *toString* method; that is, only the *toString* method can use the *gallonsFormat* object. To use the *DecimalFormat* class, we import the class on line 5. We then build the *String* to return by concatenating labels for each instance variable with the values of the instance variables. The *toString* method can be used in a client class containing the *main* method, for instance, to print *Auto* objects using a single statement instead of calling all the class accessor methods.

To implement our *equals* method (lines 114–131), we first need to check that the parameter’s type is *Auto*. The ***instanceof*** binary operator, whose left operand is an object reference and right operand is a class, evaluates to *true* if the object reference can be cast to an instance of the class (for example, if it is an object reference of that class) and *false* otherwise (**Table 7.4**). We use the *instanceof* operator at line 119 to determine if the parameter *o* can be cast to an *Auto* object reference (most likely, when sent by a client, *o* will be an *Auto* reference). If it cannot, we return *false* (line 130). Otherwise, we can proceed with comparing *o*’s fields and this *Auto* object’s fields. Before performing the comparison, we must cast the *Object* reference *o* to an *Auto* (line 119). Otherwise, there would be a compiler error when trying to access the instance variable *model* with the *Object o*, because *model* is an instance variable of class *Auto* and not of class *Object*.

The *instanceof* Operator

Operator	Syntax	Operation
instanceof	*objectReference* instanceof *ClassName*	evaluates to *true* if *objectReference* is of *ClassName* type; *false* otherwise

Description

The pattern matching feature, recently added to Java, makes it possible to test if an object reference is of a certain data type and cast it to that data type in one expression, as done on line 119.

Previously, we had to do this in two steps: first, test if the object reference (*o*) is of a particular data type (*Auto*) using the *instanceof* operator, and second, cast *o* to an *Auto* object reference (*objAuto*), as in the following code:

```
if ( o instanceof Auto )
{
   Auto objAuto = ( Auto ) o;
   // call methods of the Auto class with objAuto
```

Description

We compare each instance variable in the parameter object, *objAuto*, with the same instance variable in this object. We return *true* if the corresponding instance variables in each object have the same values; otherwise, we return *false*.

Notice that line 121 calls the *equals* method of the *String* class to compare the values of *model* in the objects because *model* is a *String* object reference. Notice also that because instance variables are in scope for methods, our *equals* method is able to directly access the instance variables of both this object and the *Auto* object, *objAuto*.

Example 7.13 puts Version 6 of the *Auto* class to work. We instantiate two objects that differ only in the model. On line 10, we explicitly call *toString* to print the fields of the *sporty* object. On line 14, we implicitly call the *toString* method; *toString* is called automatically because the *compact* object is the argument sent to the *println* method. On lines 16–19, we compare the two objects using the *equals* method and print the results. The output is shown in **Figure 7.8**.

Example 7.13 *Auto* Client, Version 6

```
/* Auto Client, version 6
   Anderson, Franceschi
*/

public class AutoClient
{
  public static void main( String [ ] args )
  {
     Auto sporty = new Auto( "Spyder", 0, 0.0 );
     System.out.println( sporty.toString( ) );

     Auto compact = new Auto( "Accent", 0, 0.0 );
     System.out.println( );
     System.out.println( compact );

     if ( compact.equals( sporty ) )
       System.out.println( "\nsporty and compact are equal" );
     else
       System.out.println( "\nsporty and compact are not equal" );
  }
}
```

Description

Figure 7.8 Output from Example 7.13

Description

Notice that the *toString* method returns a *String*. It does not output the data itself using *System.out.println*. By returning the object values as a *String*, the class is more reusable. The client can choose how best to handle the returned values. If the client is a text application, it may indeed print the values using *System.out.println*. But if the client has a graphics window, it may want to display the data in a text box or by using the JavaFX *fillText* method. Notice also that the *equals* method does not output a message stating whether the two objects are equal. Instead, the *equals* method returns *true* or *false* so that the client can test the return value and change its behavior depending on whether the objects are equal or not equal. For similar reasons, the accessors return the values of the instance variables rather than outputting those values.

SOFTWARE ENGINEERING TIP

The class methods should communicate directly with the client, and the client should handle the communications with the user.

As a general software engineering principle, class methods should communicate only with the client. The client, in turn, communicates with the user. In particular, the class methods should not output anything; the decision on what to output should be left to the client of the class. Thus, the client accepts requests from the user, calls the class methods and receives the returned values from the class, and then communicates those results to the user, as appropriate. **Figure 7.9** illustrates this design principle.

Figure 7.9 Communication Flow among the User, Client, and Class

Description

CODE IN ACTION

Within the online resources, you will find a movie with step-by-step illustrations on how to define a class. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

```
Before instantiating an Auto object:
the count of Auto objects is 0

After instantiating an Auto object:
the count of Auto objects is 1
```

7.11 *Static* Class Members

As we have mentioned, a separate set of instance variables is created for each object that is instantiated. In addition to instance variables, classes can define **class variables**, which are created only once, when the JVM initializes the class. Thus, class variables exist before any objects are instantiated, and each class has only one copy of its class variables.

You can designate a class variable by using the keyword *static* in its definition. Also, *static* variables that are constants are usually declared to be *public* because they typically are provided to make it easier to use the class. For example, the *PI* and *E static* constants in the *Math* class are provided so that our applications do not need to define those commonly used values. The *Color* class provides *public static* color constants, such as *RED,* for convenience when using common colors. Also, the maximum and minimum values for data types are made available as the *MAX_VALUE* and *MIN_VALUE public static* constants of the *Integer, Double*, and *Character* wrapper classes.

If, however, you define a *static* variable for your class that is not a constant, it is best to define it as *private* and provide accessor and mutator methods, as appropriate, for client access to the *static* variable.

We finish our *Auto* class, with Version 7, partially shown in **Example 7.14**, by defining a *private static* variable to count the number of objects that have been instantiated during the application. We call this class variable *countAutos* and initialize it to 0 (line 14). Because a constructor is called whenever an object is instantiated, we can update the count by incrementing the value of *countAutos* in the class constructors (lines 24 and 37).

Example 7.14 *Auto* Class, Version 7

```
  1 /* Auto class, Version 7
  2    Anderson, Franceschi
  3 */
  4 
  5 import java.text.DecimalFormat;
  6 
  7 public class Auto
  8 {
  9     // instance variables
 10     private String model;          //  model of auto
 11     private int milesDriven;       //  number of miles driven
 12     private double gallonsOfGas;   //  number of gallons of gas
 13 
 14     private static int countAutos = 0;  // static class variable
 15 
 16     // Constructors:
 17     //  initializes model to "unknown";
 18     //  milesDriven is autoinitialized to 0
 19     //        and gallonsOfGas to 0.0;
 20     // increments countAutos
 21     public Auto( )
 22     {
 23        model = "unknown";
 24        countAutos++;    // increment static count of Auto objects
 25     }
 26 
 27     // allows client to set beginning values for
 28     // model, milesDriven, and gallonsOfGas;
 29     // increments countAutos
 30     public Auto( String startModel,
 31                  int startMilesDriven,
 32                  double startGallonsOfGas )
 33     {
 34        model = startModel;
 35        setMilesDriven( startMilesDriven );
 36        setGallonsOfGas( startGallonsOfGas );
 37        countAutos++;    // increment static count of Auto objects
 38     }
        ...
 61     // Accessor method:
 62     // returns countAutos
 63     public static int getCountAutos( )
 64     {
 65       return countAutos;
 66     }
        ...
145 }
```

Description

When you define a *static* variable for your class, its accessor and mutator methods must be defined as ***static* methods**, also called **class methods**. To do this, insert the keyword *static* in the method headers after the access modifier. We provide a *static* accessor method for the client to get the count of *Auto* objects (lines 61–66). We do not provide a mutator method, however, because clients of the class should not be able to update the value of *countAutos.* The constructors update the count automatically.

Methods that are defined to be *static* are subject to the following important restrictions, which are summarized in **Table 7.5**:

Access Restrictions for *static* and Non-*static* Methods

	***static* Method**	**Non-*static* Method**
Access instance variables?	no	yes
Access *static* class variables?	yes	yes
Call *static* class methods?	yes	yes
Call non-*static* instance methods?	no	yes
Use the reference *this*?	no	yes

Description

- *static* methods can reference only *static* variables.
- *static* methods can call only *static* methods.
- *static* methods cannot use the object reference *this*.

Again, it makes sense that *static* methods cannot access instance variables because *static* methods are associated with the class, not with any object. Further, a *static* method can be called before any objects are instantiated, so there will be no instance variables to access. Attempting to access an instance variable *xxx* from a *static* method will generate this compiler error:

```
non-static variable xxx cannot be referenced from a static context
```

Description

Notice that the *getCountAutos* method (lines 61–66) is declared to be *static* and references only the *static countAutos* variable.

A non-*static*, or **instance**, method, on the other hand, can reference both class variables and instance variables, as well as class methods and instance methods.

Example 7.15 shows Version 7 of our *AutoClient* class. At line 11 we call the *getCountAutos* method before instantiating any objects, then in line 17 we call the *getCountAutos* method again after instantiating one object. As **Figure 7.10** shows, the *getCountAutos* method first returns 0, then 1. Notice that in both calls to the *static* method, we use the dot operator with the class name rather than an object reference.

Example 7.15 *Auto* Client, Version 7

```
 1 /* Auto Client, Version 7
 2    Anderson, Franceschi
 3 */
 4
 5 public class AutoClient
 6 {
 7   public static void main( String [ ] args )
 8   {
 9      System.out.println( "Before instantiating an Auto object:"
10                          + "\nthe count of Auto objects is "
11                          + Auto.getCountAutos( ) );
12
13      Auto sporty = new Auto( "Spyder", 0, 0.0 );
14
15      System.out.println( "\nAfter instantiating an Auto object:"
16                          + "\nthe count of Auto objects is "
17                          + Auto.getCountAutos( ) );
18   }
19 }
```

Description

Figure 7.10 Output from Example 7.15

Description

Well, there it is. We’ve finished defining our *Auto* class. Although it’s a large class, we were able to build the *Auto* class incrementally using stepwise refinement.

```
r1 = 5 / 3
r2 = 7
r3 = 0
r4 = -5 / 7
gcd( 1635, 75 ) = 15
gcd( 7 * 11 * 13, 3 * 7 * 11 ) = 77
r5 = 25 / 10
In lowest terms, r5 = 5 / 2
r6 = -2310 / 1771
In lowest terms, r6 = -30 / 23
5 / 3 + 6 / 4 = 19 / 6
5 / 3 * 6 / 4 = 5 / 2
```

7.12 Another Example: The *Rational* Class

A rational number can be represented by two integers: its numerator and its denominator. There are many operations that we can do with rational numbers: addition, subtraction, multiplication, division, reducing to lowest terms, and others. **Example 7.16** shows how we can encapsulate a rational number and these operations in a *Rational* class. It includes two instance variables storing the numerator and the denominator (lines 6 and 7), as well as many methods (the method headers are at lines 88, 101, 112) to perform some of these operations. Operations such as subtraction and division are left as exercises.

Example 7.16 The *Rational* Class

```java
/* Rational class
   Anderson, Franceschi
*/
public class Rational
{
 private int num;
 private int denom;

 /**
  * Default constructor; num is set to 0, denom is set to 1
  */
 public Rational( )
 {
   denom = 1;
 }

 /**
  * Overloaded constructor, denom is set to 1
  * @param num an int, the initial value for num
  */
 public Rational( int num )
 {
   denom = 1;
   setNum( num );
 }

 /**
  * Overloaded constructor
  * @param num an int, the initial value for num
  * @param denom an int, the initial value for denom
  */
 public Rational( int num, int denom )
 {
   this.denom = 1;
   setNum( num );
   setDenom( denom );
 }

 /**
  * Accessor for num
  * @return an int, the value of  num
  */
 public int getNum( )
 {
   return num;
 }

 /**
  * Accessor for denom
  * @return an int, the value of denom
  */
 public int getDenom( )
 {
   return denom;
 }

 /**
  * Mutator for num
  * @param num an int, the new value for num
  */
 public void setNum( int num )
 {
   this.num = num;
 }

 /**
  * Mutator for denom
  * @param denom an int, the new value for denom
  */
 public void setDenom( int denom )
 {
   if( denom != 0 )
   {
     this.denom = denom;
     if ( this.denom < 0 )
     {
       this.denom = -this.denom;
       this.num = -this.num;
     }
   }
 }

 /**
  * add method
  * @param r a Rational
  * @return a Rational, the sum of this Rational and r
  */
 public Rational add( Rational r )
 {
   Rational result = new Rational( );
   result.num = num * r.denom + denom * r.num;
   result.denom = denom * r.denom;
   return result;
 }

 /**
  * multiply method
  * @param r a Rational
  * @return a Rational, the product of this Rational and r
  */
 public Rational multiply( Rational r )
 {
   Rational result = new Rational( );
   result.num = num * r.num;
   result.denom = denom * r.denom;
   return result;
 }

 /**
  * simplify method; reduces this Rational to simplest terms
  */
 public void simplify( )
 {
   int g = Algebra.gcd( num, denom );
   num = num / g;
   denom = denom / g;
  }

 /**
  * toString method
  * @return a String, the fraction for this Rational
  */
 public String toString( )
 {
   String result = "" + num;
   if( num != 0 && denom != 1 )
     result += " / " + denom;
   return result;
 }

 /**
  * equals method
  * @param o an Object
  * @return a boolean, true if this Rational is equal to o,
  *                    false otherwise
  */
 public boolean equals( Object o )
 {
   if ( o instanceof Rational objRational )
     return num * objRational.denom == objRational.num * denom;
   else
     return false;
 }
}
```

Description

At lines 9–37, we provide three constructors. The *setDenom* method (lines 66–81) does more than set the denominator to a new value and enforce the constraint that the denominator must not be *0*. It also ensures that the denominator is always positive. This is a desirable feature, to mirror how a rational number is visualized and written in math. If the new denominator is negative (line 75), it changes the values of the numerator and denominator to their opposite values (lines 77–78). In this way, the denominator is always positive.

The *add* method is coded at lines 83–94. The *add* method only takes one parameter, a *Rational* object that we add to *this Rational* object (the object calling the *add* method in the *main*). Adding two rational numbers results in a rational number; thus, the *add* method returns a *Rational*. Inside the body of the method, we can start our code by declaring and instantiating a *Rational* object (line 90) and end our code with returning that *Rational* object (line 93). In between (lines 91–92), we define it. If the two rational numbers are `a / b` and `c / d`, adding them together results in `( a * d + b * c ) / ( b * d )` as explained below:

```
a / b + c / d = ( a * d ) / ( b * d ) + ( b * c ) / ( b * d ) =
( a * d + b * c ) / ( b * d )
```

Description

Thus, if *a / b* represents this *Rational* object and *c / d* represents the parameter *Rational* object, the equation translates into the code at lines 91–92.

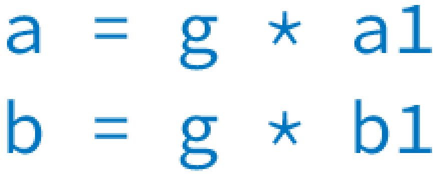
The *multiply* method is coded at lines 96–107, and implements the following equation:

```
( a / b ) * ( c / d ) = ( a * c ) / ( b * d )
```

Description

Its implementation is very similar to the *add* method: we first declare and instantiate a *Rational* object, then modify it, and finally return it.

In order to reduce a rational number to its lowest terms, we first need to calculate the greatest common divisor (gcd) between the numerator and the denominator. If the rational number is *a* / *b* and the gcd of *a* and *b* is *g*, then we have:



```
a = g * a1
b = g * b1
```

Description

where *a1* and *b1* are prime together.

Thus `a / b = ( g * a1 ) / ( g * b1 ) = a1 / b1` and is now in lowest terms.

The *simplify* method implements the above and is coded at lines 109–117. At line 114, we call the *static gcd* method of the *Algebra* class to calculate the gcd between *num* and *denom*. At lines 115 and 116, we update *num* and *denom* by dividing them by their gcd.

Calculating the gcd between two integers is independent of any *Rational* object; thus, we place that method in a new class, *Algebra* (**Example 7.17**), and declare the *gcd* method (lines 6–31) *static*. It accepts two *int* parameters and returns their gcd.

Example 7.17 The *Algebra* Class

```
 1 /* Algebra class
 2    Anderson, Franceschi
 3 */
 4 public class Algebra
 5 {
 6
 7   /**
 8    * gcd static method: computes and returns the gcd of a and b
 9    * @param a an int
10    * @param b an int
11    * @return an int, the gcd of a and b
12    */
13   public static int gcd( int a, int b )
14   {
15     // convert a and b to positive integers
16     a = Math.max( a, -a );
17     b = Math.max( b, -b );
18
19     if( b == 0 )
20       return a;
21     else
22     {
23       int r = a % b;
24       while( r != 0 )
25       {
26         a = b;
27         b = r;
28         r = a % b;
29       }
30       return b;
31     }
32   }
33 }
```

Description

At lines 15–17, we convert *a* and *b* to positive numbers, in case one of them is negative. Then, we implement the Euclid algorithm. If we divide *a* by *b*, we have: `a = b * q + r` where *q* is the quotient of the division and *r* is the remainder (`0 < = r < b`).

Euclid’s algorithm is based on the fact that the gcd of *a* and *b* is equal to the gcd of *b* and *r*. The proof is as follows:

Let’s call *g1* the gcd of *a* and *b*, and *g2* the gcd of *b* and *r*.

G1 divides *a* and *g1* divides *b*; thus, *g1* divides `a − b * q`, which is equal to *r*. Since *g1* divides *b* and *r*, *g1* is a common divisor of *b* and *r*. Thus, `g1 < = g2`, since *g2* is the greatest common divisor of *b* and *r*.

G2 divides *b* and *g2* divides *r*; thus, *g2* divides `b * q + r`, which is equal to *a*. Since *g2* divides *a* and *b*, *g2* is a common divisor of *a* and *b*. Thus, `g2 < =g1`, since *g1* is the greatest common divisor of *a* and *b*.

That means that *g1* and *g2* are equal.

In Euclid's algorithm, we keep performing the division of *a* by *b,* and we replace *a* by *b* and *b* by *r* as long as the remainder is not *0* (lines 23–28). When the remainder of the division is *0* (line 24), we have found the gcd and we exit the loop.

Here is a run of the algorithm computing the gcd of *1635* and *75*:

```
1635 = 75 * 21 + 60 (a = 1635, b = 75, q = 21, r = 60)
75 = 60 * 1 + 15 (a = 75, b = 60, q = 1, r = 15)
60 = 15 * 4 + 0 (a = 60, b = 15, q = 4, r = 0)
```

Description

Since the last value of *r* is *0*, the gcd is the last value of *b*, *15*.

Going back to Example 7.16 and the *toString* method (lines 119–129), we want to return a *String* representation of the *Rational* object that is similar to how we would write that rational number in math. For example, if the numerator is *5* and the denominator is *7*, we want to return *5 / 7*. But if the denominator is *1*, we just want to return the value of the numerator. And if the numerator is *0*, we just want to return *0*. At line 125, we initialize the *String result* with the value of *num*. If the denominator is not *1* and the numerator is not *0* (line 126), we append to *result* the */* character and *denom*.

There are many choices to be made when implementing the *Rational* class. For example, in our implementation, we never call the *simplify* method inside the *add*, *multiply*, mutators, or anywhere inside the class. We leave it to the client of the class to call *simplify* whenever appropriate. For example, a client might want to output something like:

```
10 / 5 in lowest terms is equal to 2
```

Description

With our implementation, we leave the flexibility of doing this to the client. In our *add* and *multiply* method, we also chose not to modify this *Rational*, but instead to return a new *Rational*.

Thus, a client can easily write code such as:

```
Rational r3 = r1.add( r2 ); // r1 and r2 are two existing Rationals
```

Description

And then easily output something like:

```
3 / 5 + 2 / 3 = 19 / 15
```

Description

Example 7.18 shows a client class calling the various methods of the *Rational* class. Its output is shown in **Figure 7.11**.

Example 7.18 The *RationalClient* Class

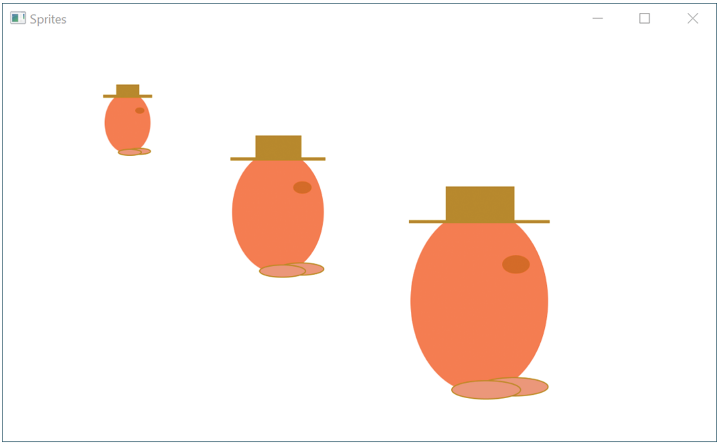
```
/* RationalClient class
   Anderson, Franceschi
*/
public class RationalClient
```

```java
/* RationalClient class
   Anderson, Franceschi
*/
public class RationalClient
{
  public static void main( String [] args )
  {
    Rational r1 = new Rational( 5, 3 );
    System.out.println( "r1 = " + r1 );
    Rational r2 = new Rational( 7 );
    System.out.println( "r2 = " + r2 );
    Rational r3 = new Rational( );
    System.out.println( "r3 = " + r3 );
    Rational r4 = new Rational( 5, -7 );
    System.out.println( "r4 = " + r4 );

    int g = Algebra.gcd( 45, 75 );
    System.out.println( "The gcd of 45 and 75 is " + g );
    g = Algebra.gcd( 3 * 7 * 11 * 10, 7 * 11 * 23 );
    System.out.println( "gcd( 7 * 11 * 13, 3 * 7 * 11 ) = " + g );

    Rational r5 = new Rational( 25, 10 );
    System.out.println( "r5 = " + r5 );
    r5.simplify( );
    System.out.println( "In lowest terms, r5 = " + r5 );

    Rational r6 = new Rational( 3 * 7 * 11 * 10, -7 * 11 * 23 );
    System.out.println( "r6 = " + r6 );
    r6.simplify( );
    System.out.println( "In lowest terms, r6 = " + r6 );

    Rational r7 = new Rational( 5, 3 );
    Rational r8 = new Rational( 6, 4 );
    Rational r9 = r7.add( r8 );
    r9.simplify( );
    System.out.println( r7 + " + " + r8 + " = " + r9 );

    Rational r10 = r7.multiply( r8 );
    r10.simplify( );
    System.out.println( r7 + " * " + r8 + " = " + r10 );
  }
}
```

Description

Figure 7.11 Output of *RationalClient*

Description



7.13 Graphical Objects

Now that we know how to design our own classes, we can separate a graphical object from an application. For example, we can define a sprite in its own *Sprite* class and write a client application that will instantiate and draw *Sprite* objects. This will allow us to encapsulate the sprite's data and the code for drawing the sprite within the *Sprite* class. It also promotes reuse of the *Sprite* class by other programmers who might want to create *Sprite* objects for different applications.

The *Sprite* class is shown in **Example 7.19**. We start by defining the instance variables. We need the starting (*x*, *y*) coordinate to draw the sprite, so we define two *int* instance variables to hold those values (lines 9–10). The *sX* variable is the leftmost point of the drawing (the upper-left corner of the rectangle enclosing the body), and *sY* represents the topmost point (the top of the sprite's hat).

Example 7.19 The *Sprite* Class

```java
/* Sprite class
   Anderson, Franceschi
*/
import javafx.scene.canvas.*;
import javafx.scene.paint.*;

public class Sprite
{
   private int sX;
   private int sY;
   private double scale;

   /** default constructor
   *  sX = sY = 0; scale is set to 1
   */
   public Sprite( )
   {
     scale = 1;
   }

   /* overloaded constructor
      accepts values for starting x and y coordinates
      and scale
   */
   public Sprite( int sX, int sY, double scale )
   {
      setCoordinates( sX, sY );
      setScale( scale );
   }

   /* setCoordinates
   * accepts new values for starting x and y;
   * returns a reference to this object
   */
   public Sprite setCoordinates( int sX, int sY )
   {
      this.sX = sX;
      this.sY = sY;
      return this;
   }

   /* mutator for scale
   * returns a reference to this object
   */
   public Sprite setScale( double scale )
   {
      this.scale = ( scale > 0 ? scale : this.scale );
      return this;
   }

   /* draw method
   *  draws Sprite at current sX and sY
   *  multiplying lengths by scale
   *  accepts GraphicsContext for canvas
   */
   public void draw( GraphicsContext gc )
   {
       gc.setFill( Color.CORAL ); // body
       gc.fillOval( sX, sY + 15 * scale, 90 * scale, 120 * scale );
       gc.setFill( Color.DARKGOLDENROD ); // hat
       gc.fillRect( sX + 23 * scale, sY, 45 * scale, 22 * scale );
       gc.setStroke( Color.DARKGOLDENROD ); // hat brim
       gc.setLineWidth( 3 );
       gc.strokeLine( sX, sY + 23 * scale,
                      sX + 90 * scale, sY + 23 * scale );
       gc.setFill( Color.CHOCOLATE ); // eye
       gc.fillOval( sX + 60 * scale, sY + 45 * scale,
                    18 * scale, 12 * scale );
       gc.setFill( Color.DARKSALMON ); // feet
       gc.setLineWidth( 1 );
       gc.fillOval( sX + 45 * scale, sY + 125 * scale,
                    45 * scale, 12 * scale );
       gc.strokeOval( sX + 45 * scale, sY + 125 * scale,
                      45 * scale, 12 * scale );
       gc.fillOval( sX + 27 * scale, sY + 127 * scale,
                    45 * scale, 12 * scale );
       gc.strokeOval( sX + 27 * scale, sY + 127 * scale,
                      45 * scale, 12 * scale );
   }
}
```

Description

In addition, we added one more instance variable, *scale* (line 11), to allow the client to draw sprites of different sizes. For example, a scaling factor of 1.0 will draw the sprite at full size, 0.5 will draw the sprite at half size, and 2.0 will draw a double-sized sprite.

We provide a default constructor (lines 13–19) that autoinitializes *sX* and *sY* to 0 and sets *scale* to 1. Our overloaded constructor (lines 21–29) accepts values for the three instance variables and passes those values to the appropriate mutator methods.

We provide one mutator method to change both *x* and *y* values (lines 31–40), as well as another mutator to change the scaling factor (lines 42–49).

We provide a method, *draw* (lines 51–79), that draws the sprite on a canvas. Because the sprite is drawn using methods of the *GraphicsContext* class, the application client needs to pass its *GraphicsContext* object as an argument to the *draw* method. In the *draw* method, we draw the sprite, multiplying any length measurement by the scaling factor.

We do not provide accessor methods, a *toString* method, or an *equals* method for the *Sprite* class. For a graphical object, these methods are less useful, given that the major purpose of graphical objects is to be drawn.

Now we can create the client application, *SpriteClient.java*, which is shown in **Example 7.20**. An advantage to separating the *Sprite* class from the *SpriteClient* class is that we can now easily draw two or more sprites with different sizes in different locations. The *SpriteClient* class defines three *Sprite* references as instance variables (line 11). We instantiate these *Sprite* objects with different starting coordinates and scales (lines 18–20). We call the *draw* method for each *Sprite* object (lines 22–24), passing to the *draw* method the *GraphicsContext* object reference *gc* that was returned by the *JIGraphicsUtility*'s *setUpGraphics* method (lines 16–17). The application window is shown in **Figure 7.12**.

Example 7.20 The *SpriteClient* Class

```
/* A client for the Sprite class
   Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.scene.canvas.GraphicsContext;
import javafx.stage.Stage;

public class SpriteClient extends Application
{
    private Sprite s1, s2, s3;

    @Override
    public void start( Stage stage )
    {
        GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
                  stage, "Sprites", 700, 400 );
        s1 = new Sprite( 100, 50, .5 );
        s2 = new Sprite( 225, 100, 1 );
        s3 = new Sprite( ).setCoordinates( 400, 150 ).setScale( 1.5 );

        s1.draw( gc );
        s2.draw( gc );
        s3.draw( gc );
    }

    public static void main( String [ ] args )
    {
        launch( args );
    }
}
```

Description

Figure 7.12 The *SpriteClient* Window

Description

```
Comparing objects using equals
WED does not equal FRI

Comparing objects using compareTo
WED is less than FRI

Getting the ordinal value
The value of WED is 3

Converting a String to an object
The value of day is MON
```

```
Enter a day
(Sun, Mon, Tue, Wed, Thur, Fri, Sat) > Fri
The special for FRI is fish fillet
```

7.14 Enumeration Types

Enumeration types are designed to increase the readability of programs. The enumeration type *enum* is a special kind of class declaration. It allows us to define a set of named constant objects that can be used instead of numbers in a program.

Enum types are useful for managing ordered sets where each member of the set has a name. Examples are the days of the week, months of the year, and playing cards. To represent these sets in a program, we often use numbers, such as 1 through 7 for days of the week or 1 through 12 for months of the year. The problem is that to input or output these values, we need to convert between our internal numeric representation (for example, 1–7) and the words that users recognize (Sunday, Monday, Tuesday, etc.). The *enum* type allows us to instantiate a constant object for each value in a set. The set of objects will be ordered so that we can refer to the objects by name, without the need for using numbers.

The *enum* functionality is built into *java.lang*, so we can define *enum* types without using an *import* statement.

The syntax for creating a set of *enum* objects is

```
enum EnumName { obj1, obj2, . . . };
where obj1, obj2, etc. are names for the constant objects.
```

Description

For example, the following statement defines an *enum* type to represent the days of the week:

```
enum Days { SUN, MON, TUE, WED, THUR, FRI, SAT };
```

Description

When that statement is executed, an object is instantiated for each name in the list. Each name in the list, therefore, is a reference to an object of the *enum* type *Days*.

Note that the values in the initialization list are object references (*SUN*), not *String* literals ("*SUN*").

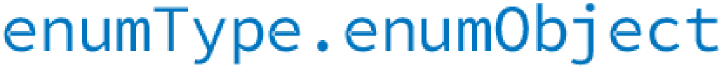
COMMON ERROR TRAP

Do not use *String* literals in the initialization list for *enum* types.

Each object has an instance variable that holds a numeric value, which is determined by its position in the list of *enum* objects. By default, the first object has the value 0, the second object has the value 1, and so on. Because the objects are an ordered set, for example, the object *THUR* is higher in value than *WED*. We can use the *enum* objects, however, without relying on the specific value of each object.

The *enum* objects are instantiated as constant objects, meaning that their values cannot be changed.

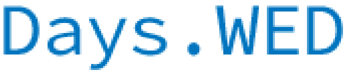
To refer to any of the constant objects in an *enum* type, we use the following dot syntax:



```
enumType.enumObject
```

Description

Thus, to refer to the *WED* object in our *Days enum* type, we use this syntax:



```
Days.WED
```

Description

Once we have defined an *enum* type, we can declare an object reference of that type. For example, the following statement defines a *Days* object reference *d*:

```
Days d;
```

Like any other object reference, the value of *d* will be *null* initially. To assign a value to the reference *d*—for example, *THUR*—we use the following statement:

```
d = Days.THUR;
```

Table 7.6 lists some useful methods that can be called with *enum* objects, and **Example 7.21** demonstrates the use of these methods.

Useful Methods for *enum* Objects

Useful Methods for *enum* Objects

Return type	**Method name and argument list**
`int`	`compareTo( Enum eObj )` compares two *enum* objects and returns a negative number if *this* object is less than the argument, a positive number if *this* object is greater than the argument, and 0 if the two objects are the same
`boolean`	`equals( Object eObj )` returns *true* if this object is equal to the argument *eObj*; returns *false* otherwise
`int`	`ordinal( )` returns the numeric value of the *enum* object; by default, the value of the first object in the list is 0, the value of the second object is 1, and

	so on
`String`	`toString( )` returns the name of the *enum* constant
`enum`	`valueOf( String enumName )` *static* method that returns the *enum* object whose name is the same as the *String* argument *enumName*

Description

Example 7.21 A Demonstration of *enum* Methods

```
 1 /* Demonstration of enum
 2    Anderson, Franceschi
 3 */
 4
 5 public class EnumDemo
 6 {
 7   public enum Days { SUN, MON, TUE, WED, THUR, FRI, SAT };
 8
 9   public static void main( String [ ] args )
10   {
11      Days d1, d2;  // declare two Days object references
12
13      d1 = Days.WED;
14      d2 = Days.FRI;
15
16      System.out.println( "Comparing objects using equals" );
17      if ( d1.equals( d2 ) )
18        System.out.println( d1 + " equals " + d2 );
19      else
20        System.out.println( d1 + " does not equal " + d2 );
21
22      System.out.println( "\nComparing objects using compareTo" );
23      if ( d1.compareTo( d2 ) > 0 )
24        System.out.println( d1 + " is greater than " + d2 );
25      else if ( d1.compareTo( d2 ) < 0 )
26        System.out.println( d1 + " is less than " + d2 );
27      else
28        System.out.println( d1 + " is equal to " + d2 );
29
30      System.out.println( "\nGetting the  ordinal value" );
31      System.out.println( "The value of " + d1 + " is "
32                       + d1.ordinal( ) );
33
34      System.out.println( "\nConverting a String to an object" );
35      Days day = Days.valueOf( "MON" );
36      System.out.println( "The value of day is " + day );
37   }
38 }
```

Description

Line 7 defines the *enum* type *Days*; this instantiates the seven constant objects representing the days of the week. On line 11, we declare two object references of the *Days enum* type. Then on lines 13 and 14, we assign *d1* a reference to the *WED* object, and we assign *d2* a reference to the *FRI* object.

Line 17 compares *d1* and *d2* using the *equals* method. Because *WED* and *FRI* are different objects, the *equals* method returns *false*. Lines 18 and 20 implicitly call the *toString* method, which prints the name of the objects.

Lines 23 and 25 call the *compareTo* method, which returns a negative number, indicating that *WED* is lower in value than

FRI.

We then retrieve the value of the *d1* object by calling the *ordinal* method (lines 31–32), which returns 3 because *WED* is the fourth object in the *enum* list.

Finally, line 35 converts from a *String* to an *enum* object using the *valueOf* method. Notice that the *valueOf* method is *static*, so we call it using our *enum* type, *Days*.

If the *String* passed to the *valueOf* method is not a name in our set of defined *enum* objects, the *valueOf* method generates an *IllegalArgumentException*.

The output from Example 7.21 is shown in **Figure 7.13**.

Figure 7.13 Output from Example 7.21

Description

We can use *enum* objects in *switch* statements to make the *case* constants more meaningful, which in turn makes the code more readable. **Example 7.22** uses our *Days enum* class to display the daily specials offered in the cafeteria.

Example 7.22 *DailySpecials* Class

```
 1 /** Specials of the Day
 2     Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6
 7 public class DailySpecials
 8 {
 9   public enum Days { SUN, MON, TUE, WED, THUR, FRI, SAT };
10
11   public static void main( String [ ] args )
12   {
13     Scanner scan = new Scanner( System.in );
14
15     System.out.print( "Enter a day\n"
16                     + "(Sun, Mon, Tue, Wed, Thur, Fri, Sat) > " );
17     String inputDay = scan.next( ).toUpperCase( );
18     Days day = Days.valueOf( inputDay );
19
20     switch ( day )
21     {
22       case MON,WED -> System.out.println( "The special for "
23                                  + day + " is barbeque chicken" );
24       case TUE,THUR -> System.out.println( "The special for "
25                                  + day + " is tacos" );
26       case FRI -> System.out.println( "The special for "
27                                  + day + " is fish fillet" );
28       case SUN,SAT -> System.out.println( "Sorry, we're closed on "
29                                  + day );
30     }
31   }
32 }
```

Description

Figure 7.14 shows the output from Example 7.22 when the user enters *Fri*.

Figure 7.14 Output from *DailySpecials*

Description

In the *DailySpecials* program, we prompt the user for a day (lines 15–17), then read the *String* entered by the user and attempt to convert it to an *enum* object by calling the *valueOf* method at line 18.

Once we have a valid *enum* value, we can use it as a *switch* variable (line 20).

Notice that we use each *enum* object name in a *case* label without qualifying it with the *Days* type. Including the *enum* type in a *switch* statement generates the following compiler error:

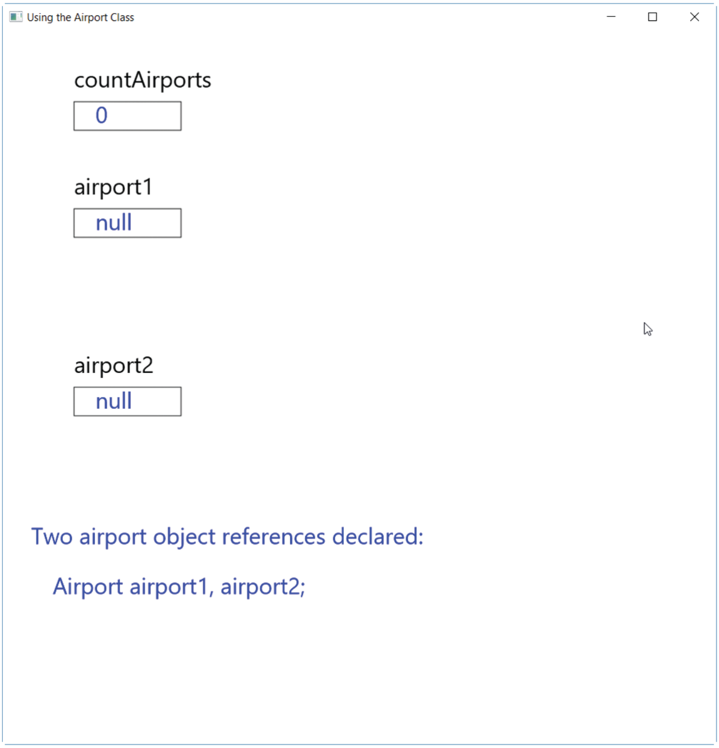
```
an enum switch case label must be the unqualified name of an enumeration constant
```

Description

Notice also that if the user enters a *String* that does not match one of the seven valid day values, the call to the *valueOf* method at line 18 will cause an *IllegalArgumentException* to occur at run time. And finally, notice that we do not need a *default* case in our *switch* statement because all our cases are exhaustive.

with these end-of-chapter questions

7.19.1	Multiple Choice Exercises Questions 14, 15, 16
7.19.2	Reading and Understanding Code Questions 21, 22, 23, 25, 27
7.19.3	Fill In the Code Questions 29, 34, 35
7.19.4	Identifying Errors in Code Questions 40, 41, 42, 44, 46
7.19.5	Debugging Area Questions 50, 51, 53, 54
7.19.6	Write a Short Program Questions 55, 56, 57, 58, 59, 60, 61, 62, 63, 64
7.19.8	Technical Writing Question 76



countAirports
0
airport1
null
airport2
null

Two airport object references declared:

Airport airport1, airport2;

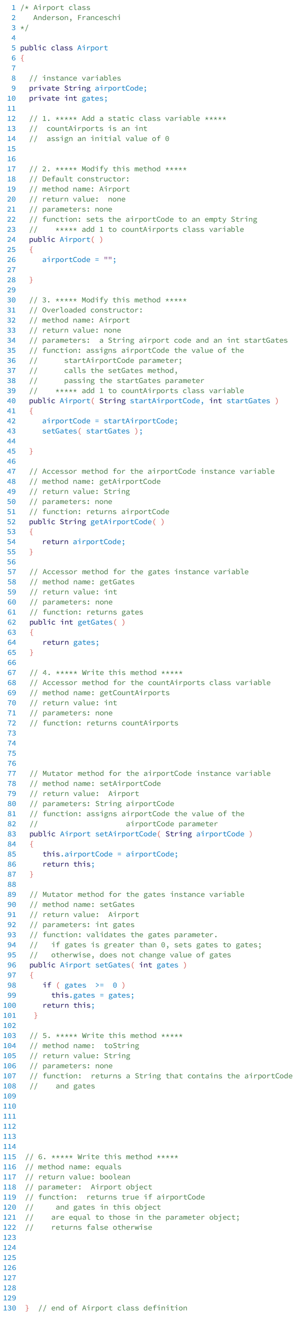
7.15 Programming Activity 2: Writing a Class Definition, Part 2

In this programming activity, you will complete the definition of the *Airport* class. Then you will run a prewritten client program that instantiates several *Airport* objects, calls the methods that you have written, and displays the values of the objects' data.

Copy into a folder on your computer all the files from this chapter's Programming Activity 2 folder in the supplied code files. Note that all files should be in the same folder.

Load the *Airport.java* source file; you'll notice that the class already contains the class definition from Programming Activity 1. Your job is to complete the class definition by adding a *static* class variable (and its supporting code) and writing the *toString* and *equals* methods. It is important to define the *static* class variable and the methods exactly as described in the comments, because the *AirportDrawing* class will call each method to test its implementation. Searching for five asterisks in a row (*****) will position you at the six places in the class definition where you will add your code. The *Airport.java* code is shown here in **Example 7.23**.

Example 7.23 The *Airport.java* File



```
/* Airport class
   Anderson, Franceschi
*/

public class Airport
{

  // instance variables
  private String airportCode;
  private int gates;

  // 1. ***** Add a static class variable *****
  //  countAirports is an int
  //  assign an initial value of 0


  // 2. ***** Modify this method *****
  // Default constructor:
  // method name: Airport
  // return value:  none
  // parameters: none
  // function: sets the airportCode to an empty String
  //    ***** add 1 to countAirports class variable
  public Airport( )
  {
     airportCode = "";

  }

  // 3. ***** Modify this method *****
  // Overloaded constructor:
  // method name: Airport
  // return value: none
  // parameters:  a String airport code and an int startGates
  // function: assigns airportCode the value of the
  //      startAirportCode parameter;
  //      calls the setGates method,
  //      passing the startGates parameter
  //    ***** add 1 to countAirports class variable
  public Airport( String startAirportCode, int startGates )
  {
     airportCode = startAirportCode;
     setGates( startGates );

  }

  // Accessor method for the airportCode instance variable
  // method name: getAirportCode
  // return value: String
  // parameters: none
  // function: returns airportCode
  public String getAirportCode( )
  {
     return airportCode;
  }

  // Accessor method for the gates instance variable
  // method name: getGates
  // return value: int
  // parameters: none
  // function: returns gates
  public int getGates( )
  {
     return gates;
  }

  // 4. ***** Write this method *****
  // Accessor method for the countAirports class variable
  // method name: getCountAirports
  // return value: int
  // parameters: none
  // function: returns countAirports




  // Mutator method for the airportCode instance variable
  // method name: setAirportCode
  // return value:  Airport
  // parameters: String airportCode
  // function: assigns airportCode the value of the
  //                    airportCode parameter
  public Airport setAirportCode( String airportCode )
  {
     this.airportCode = airportCode;
     return this;
  }

  // Mutator method for the gates instance variable
  // method name: setGates
  // return value:  Airport
  // parameters: int gates
  // function: validates the gates parameter.
  //   if gates is greater than 0, sets gates to gates;
  //   otherwise, does not change value of gates
  public Airport setGates( int gates )
  {
     if ( gates  >=  0 )
       this.gates = gates;
     return this;
   }

  // 5. ***** Write this method *****
  // method name:  toString
  // return value: String
  // parameters: none
  // function:  returns a String that contains the airportCode
  //    and gates







 // 6. ***** Write this method *****
 // method name: equals
 // return value: boolean
 // parameter:  Airport object
 // function:  returns true if airportCode
 //     and gates in this object
 //    are equal to those in the parameter object;
 //    returns false otherwise







 }  // end of Airport class definition
```

Description

When you finish modifying the *Airport* class, compile the source file. When *Airport.java* compiles without any errors, load and compile the *AirportPractice2Application.java* file. This source file contains *main*, so you will execute the application from this file. When the application begins, you should see the window shown in **Figure 7.15**.

Figure 7.15 *AirportPractice2Application* Opening Window

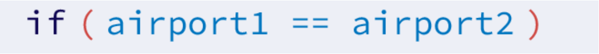
Description

As you can see, the client class has declared two *Airport* object references, *airport1* and *airport2*. The references are *null* because no *Airport* objects have been instantiated. Note also that the value of the *countAirports* class variable is displayed.

The client class will call methods of the *Airport* class to instantiate the two *Airport* objects, call the *toString* and *equals* methods, and get the value of the *static* class variable, *countAirports*. As the application does its work, it displays a status message at the bottom of the window indicating which method has been called, and it also displays the current state of the *Airport* objects. You can check your work by comparing the state of the objects with the status message.

DISCUSSION QUESTIONS

1. Explain why the *countAirports* class variable has a value of 0 before any *Airport* objects have been instantiated.
2. How does a client call the *getCountAirports* method?
3. Explain why when directly comparing the object references *airport1* and *airport2* using the following *if* statement:

   ```
   if ( airport1 == airport2 )
   ```

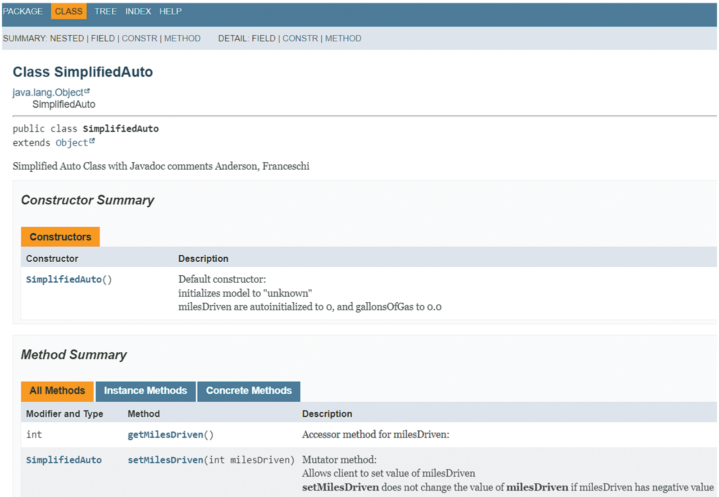
   Description

   the condition evaluates to *false*?

7.16 Creating Packages

As we have mentioned, one of the advantages of a well-written class is that it can be reused. Ideally, as you write programs, you will look for functionality that is common to many programs. It is a good practice to encapsulate this functionality into a class so that you can reuse that code in future programs. Java provides the concept of a package for easily reusing classes.

A **package** is a collection of related classes that can be imported into programs. We have imported classes from multiple Java packages: *java.util*, *java.text*, and others. We can also create our own packages, which allows us to reuse a class without needing to physically store that class in the same folder as our other source files. Instead, we create the package and *import* the class from that package into our source file. For instructions on how to create a package, please see our additional resources online.


PACKAGE
CLASS
TREE
INDEX
HELP
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

Class SimplifiedAuto

java.lang.Object
SimplifiedAuto

```
public class SimplifiedAuto
extends Object
```

Simplified Auto Class with Javadoc comments Anderson, Franceschi

Constructor Summary

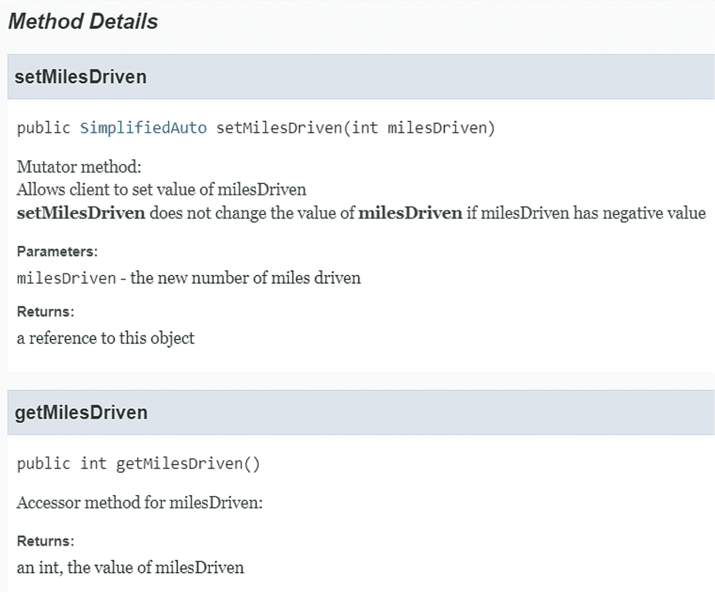
Constructors

Constructor	Description
`SimplifiedAuto()`	Default constructor: initializes model to "unknown" milesDriven are autoinitialized to 0, and gallonsOfGas to 0.0

Method Summary

All Methods
Instance Methods
Concrete Methods

Modifier and Type	Method	Description
`int`	`getMilesDriven()`	Accessor method for milesDriven:
`SimplifiedAuto`	`setMilesDriven(int milesDriven)`	Mutator method: Allows client to set value of milesDriven **setMilesDriven** does not change the value of **milesDriven** if milesDriven has negative value



Method Details

setMilesDriven

```
public SimplifiedAuto setMilesDriven(int milesDriven)
```

Mutator method:
Allows client to set value of milesDriven
setMilesDriven does not change the value of **milesDriven** if milesDriven has negative value

Parameters:

`milesDriven` - the new number of miles driven

Returns:

a reference to this object

getMilesDriven

```
public int getMilesDriven()
```

Accessor method for milesDriven:

Returns:

an int, the value of milesDriven

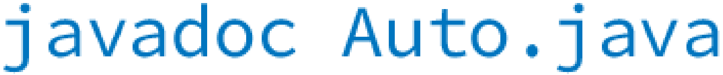
7.17 Generating Web-Style Documentation with Javadoc

In most corporations and organizations, programmers share code and frequently use classes developed by another programmer. If the class is well designed and well documented, it will be easy for others to use that class. After all, that is essentially what we have been doing by using existing Java classes. It has been easy to understand what functions these existing classes perform, what they encapsulate, how the constructors work, what the methods do, and how to use the classes. The reason that these classes are easy to understand and use is not only that they are well designed and written, but also that the available documentation, particularly on Oracle's Java website, is clear, easy to understand, complete, and represents these classes well.

We, too, will learn how to produce HTML-based documentation similar to the documentation available on Oracle's Java website.

There is a tool called **Javadoc**, provided in the Java Development Kit (JDK), to do just that. Javadoc is an executable program (actually *javadoc.exe*) located in the *bin* folder. It is invoked much the same way as the *javac* compiler, except that instead of creating *.class* files, it creates *.html* files that document the class.

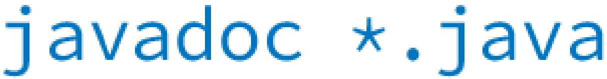
For instance, to generate documentation for our *Auto* class, we would type the following at the command line:



```
javadoc Auto.java
```

Description

If we want to generate documentation for all the source files in the directory, we would type:



```
javadoc *.java
```

Description

Your IDE may provide easy access to Javadoc through its menus.

Table 7.7 shows some of the files generated for the *Auto* class.

Selected HTML Files Generated by Javadoc

File Name	**Short Description**
Auto.html	*Auto* class documentation
allclasses-index.html	List of the classes with links
index.html	*Auto* class documentation (with frames)
package-summary.html	Class hierarchy
script	Javascript file
stylesheet.css	Style sheet

Description

If you double-click on *index.html*, you will open a webpage with the same look as the ones on Oracle's Java website.

REFERENCE POINT

The full documentation for using Javadoc can be found at *www.oracle.com/technetwork/java*.

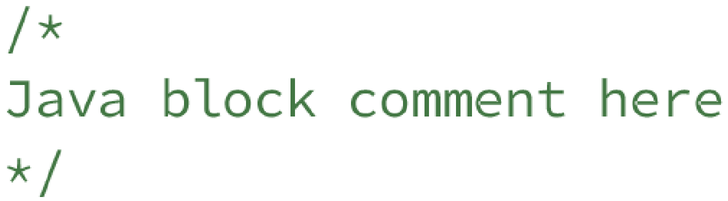
We will review a few basic Javadoc features here. Full documentation on Javadoc is available on Oracle's website.

To write comments that will be included in the Javadoc documentation, we use a special form of block comment ahead of any class, field, constructor, or method. The syntax for including Javadoc comments follows.

```
/**
Javadoc comment here
*/
```

Description

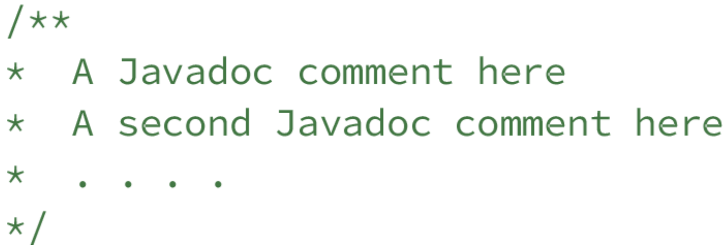
As we already know, the syntax for a Java block comment is



```
/*
Java block comment here
*/
```

Description

A Javadoc comment is just a special Java block comment. The *javac* compiler will simply ignore it, but the Javadoc executable will look for it and generate the appropriate documentation. Javadoc discards all white space characters and the * at the beginning of each line until a character other than white space or * is encountered. The industry convention is to start every line of a Javadoc comment with an *. Therefore, we recommend the following syntax:



```
/**
*  A Javadoc comment here
*  A second Javadoc comment here
*  . . . .
*/
```

Description

Class documentation comprises two parts:

- A description section
- A tag section

SOFTWARE ENGINEERING TIP

When coding a documentation block, use an * at the beginning of each line to indicate that this is a documentation comment.

Javadoc recognizes two types of tags: block tags and inline tags. We will discuss block tags only.

Block tags start with the character @. **Table 7.8** lists two block tags, *@param* and *@return*, along with an explanation of each.

Selected Javadoc Tags

Tag	Most Common Syntax	Explanation
`@param`	`@param variableName description`	Adds a parameter to the parameter section
`@return`	`@return text`	Adds a description for the return type

Description

In the description section and inside the tag section, the text should be written in HTML; therefore, HTML tags such as <br> (break) or (bold) can be used. The tag <br> inserts a new line; the tag will change the text style to bold until the end tag is encountered. Starting with Java 9, Javadoc supports HTML5.

Example 7.24 shows a simplified version of our *Auto* class incorporating some documentation comments.

Example 7.24 The *SimplifiedAuto* Class

```
/**  Simplified Auto Class with Javadoc comments
     Anderson, Franceschi
*/

public class SimplifiedAuto
{
 private String model;
 private int milesDriven;
 private double gallonsOfGas;

 /**
 * Default constructor:<br>
 * initializes model to "unknown"<br>
 * milesDriven are autoinitialized to 0, and gallonsOfGas to 0.0
 */
 public SimplifiedAuto( )
 {
    model = "unknown";
 }

 /**
 * Mutator method:<br>
 * Allows client to set value of milesDriven<br>
 * <strong>setMilesDriven</strong> does not change the value
 * of <strong>milesDriven</strong> if newMilesDriven has negative value
 * @param milesDriven the new number of miles driven
 * @return a reference to this object
 */
 public SimplifiedAuto setMilesDriven( int milesDriven )
 {
    if ( milesDriven > 0 )
        this.milesDriven = milesDriven;
    return this;
 }

 /**
 * Accessor method for milesDriven:<br>
 * @return an int, the value of milesDriven
 */
 public int getMilesDriven( )
 {
    return milesDriven;
 }
}
```

Description

SOFTWARE ENGINEERING TIP

When you write a class, add a few documentation comments and generate the web-style documentation. Show the webpages to friends or colleagues and ask them if they fully understand what the class encapsulates and what it is about. Ask them a few questions about the constructor and the methods. This is a good way to check if your class is well designed and ready for reuse.

Figure 7.16 shows part of the generated *index.html* file, and

Figure 7.17 shows the generated documentation for the *setMilesDriven* and *getMilesDriven* methods.

Figure 7.16 *SimplifiedAuto* Class Web-Style Documentation

Figure 7.17 Web-Style Documentation for the Mutator and Accessor Methods

Description

© Lois GoBe/Shutterstock.

CHAPTER REVIEW

7.18 Chapter Summary

- The members of a Java class include its instance variables, class variables, and methods.
- One copy of each instance variable is created for every object instantiated from the class. One copy of each class variable and method is shared by all objects of the class.
- By convention, class names are nouns and begin with a capital letter; all internal words begin with a capital letter, and other letters are lowercase. Method names are verbs and begin with a lowercase letter; internal words begin with a capital letter, and all other letters are lowercase. Nonconstant instance variables are nouns and follow the same capitalization rules as methods. Constant fields have all capital letters with internal words separated by an underscore.
- The *public* access modifier allows the class or member to be accessed by other classes. The *private* access modifier specifies that the class or member can be accessed only by other members of the same class. Package access allows other classes in the same package or folder to access the class or class members.
- Classes, constructors, *final* class variables, and class methods typically are declared as *public*, and instance variables typically are declared as *private*.

- Instance variables reflect the properties that all objects will have in common. Instance variables are defined by specifying an access modifier, data type, identifier, and, optionally, an initial value. Instance variables can be declared to be *final*.
- A method is defined by providing a method header, which specifies the access modifier, a return type, the method name, and a parameter list. The method body is enclosed in curly braces. Value-returning methods return the result of the method using one or more *return* statements. A method with a *void* return type does not return a value.
- Instance variables and methods have class scope in that they can be accessed anywhere in the class.
- A method can reference the instance variables of its class, the parameters sent to the method, and local variables declared by the method, and it can call other methods of its class.
- A method can be overloaded by defining another method with the same name but a different signature; that is, with a different number of parameters or with parameters of different data types.
- Constructors are responsible for initializing the instance variables of the class.
- If we don't provide a constructor, the compiler provides a default constructor, which is a constructor that takes no arguments. This default constructor assigns default initial values to all the instance variables. Numeric variables are given the value of 0, characters are given the value of the Unicode null character, *boolean* variables are given the value of *false*, and object references are given the value of *null*. Local variables declared in methods are not given initial values automatically.
- Accessor methods are named *getIV*, where *IV* is an instance variable name; the return data type is the same as the instance variable, and the body of the method simply returns the value of the instance variable.
- Mutator methods are named *setIV*, where *IV* is an instance variable name; the return data type is *void* or a reference to

this object, and the method takes one argument, which is the same data type as the instance variable and contains the new value for the instance variable. The body of the method should validate the new value and, if the new value is valid, assign the new value to the instance variable.

- When a method begins executing, the JVM sets the object reference *this* to refer to the object for which the method has been called.
- The *toString* method is called automatically when an object reference is used as a *String*, and its job is to provide a printable representation of the object data.
- The *equals* method compares two objects for equality; that is, it should return *true* only if the corresponding instance variables in both objects are equal in value, and *false* otherwise.
- *Static* class variables are created when the class is initialized. Thus, class variables exist before any objects are instantiated, and each class has only one copy of the class variables. *Static* variables that are constants are usually declared to be *public* because they typically are provided to allow the client to set preferences for the operations of a class.
- *Static* class methods can reference only *static* variables, can call only *static* methods, and cannot use the object reference *this*.
- A non-*static*, or instance, method can reference both class and instance variables, as well as class and instance methods, and the reference *this*.
- A graphical object usually has instance variables for the starting (*x*, *y*) coordinate. It also provides a *draw* method that takes a *GraphicsContext* object as a parameter and includes the code to draw the graphical object.
- Enumeration types can be defined to give meaning to ordered sets that are represented in a program by numbers. For each name in an *enum* type initialization list, a constant object is created with an instance variable having a sequential numeric value. References can be defined of the *enum* type. Objects of the *enum* type can be

compared, printed, and requested to return their numeric value.

- Javadoc, which is part of the JDK, generates documentation for classes. To use Javadoc, you enclose a description of each class, method, and field in a block comment beginning with /** and ending with */. In addition, you can describe each parameter using the *@param* tag and return value using the *@return* tag.

7.19 Exercises, Problems, and Projects

7.19.1 Multiple Choice Exercises

1. What can you say about the name of a class?

    - It must start with an uppercase letter.
    - The convention is to start with an uppercase letter.

4. What can you say about the name of constructors?

    - They must be the same name as the class name.
    - They can be any name, just like other methods.

7. What is a constructor's return type?

    - *void*
    - *Object*
    - The class name
    - A constructor does not have a return type.

12. It is legal to have more than one constructor in a given class.

    - true
    - false

15. In a class, if a field is *private*,

    - it can be accessed directly from any class.
    - it can be accessed directly only from inside its class.

18. In a typical class, what is the general recommendation for access modifiers?

- Instance variables are *private* and methods are *private*.
- Instance variables are *private* and methods are *public*.
- Instance variables are *public* and methods are *private*.
- Instance variables are *public* and methods are *public*.

23. In a class, fields

- can only be basic data types.
- can only be basic data types or existing Java types (from existing classes).
- can be basic data types, existing Java types, or user-defined types (from user-defined classes).

27. Accessors and mutators are

- instance variables of a class.
- used to access and modify field variables of a class from outside the class.
- constructor methods.

31. Accessor methods typically take

- no parameter.
- one parameter, of the same type as the corresponding field.

34. Mutator methods typically take

- no parameter.
- one parameter, of the same type as the corresponding field.

37. Accessor methods typically

- are *void* methods.
- return the same type as the corresponding field.

40. To enable method chaining, mutator methods

    - return a reference to *this* object.
    - return the same type as the corresponding field.

43. When coding a method that performs calculations on fields of that class,

    - these fields must be passed as parameters to the method.
    - these fields do not need to be passed as parameters to the methods because the class methods have direct access to them.

46. What is the keyword used for declaring a constant?

    - *static*
    - *final*
    - *constant*

50. What is the keyword used for declaring a class variable or method?

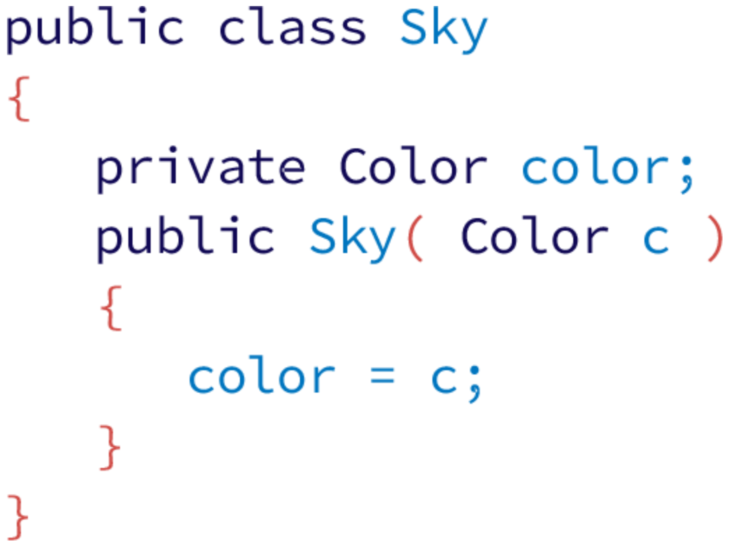
    - *static*
    - *final*
    - *class*

54. What can you say about *enum*?

    - It is part of the package *java.lang*.
    - It can be used for self-documentation, improving the readability of your code.
    - An *enum* object is a constant object.
    - All of the above.

7.19.2 Reading and Understanding Code

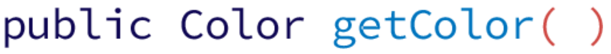
For Questions 17 and 18, consider that inside the class *Sky*, we have already coded the following:



```
public class Sky
{
    private Color color;
    public Sky( Color c )
    {
        color = c;
    }
}
```

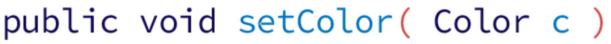
Description

1. Consider the following method header:

   ```
   public Color getColor( )
   ```

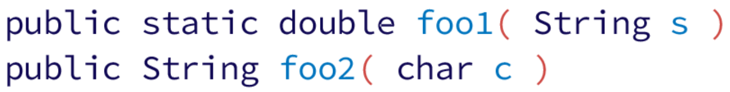
   Description
   Is this method a constructor, mutator, or accessor?
2. Consider the following method header:

   ```
   public void setColor( Color c )
   ```

   Description
   Is this method a constructor, mutator, or accessor?

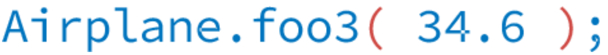
For Questions 19 through 24, consider that the class *Airplane* has two methods with the following method headers; we also have a default constructor already coded.



```
public static double foo1( String s )
public String foo2( char c )
```

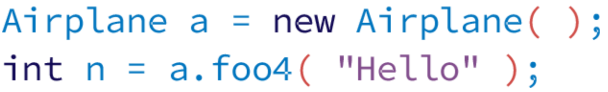
Description

1. What is the return type of method *foo1*?
2. What is the return type of method *foo2*?
3. Is method *foo1* a class or instance method? Explain.
4. Is method *foo2* a class or instance method? Explain.
5. Write a line or two of code to call method *foo1* from a client class.
6. Write a line or two of code to call method *foo2* from a client class. Assume we have instantiated an object named *a1*.
7. Inside method *main*, we see code like

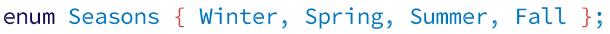
   ```
   Airplane.foo3( 34.6 );
   ```

   Description
   From this, reconstruct the header of method *foo3* (which belongs to the class *Airplane*); make appropriate assumptions if necessary.
8. Inside method *main*, we see code like

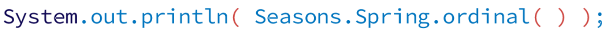
   ```
   Airplane a = new Airplane( );
   int n = a.foo4( "Hello" );
   ```

   Description
   From this, reconstruct the header of method *foo4* (which belongs to class *Airplane*).
9. If you have defined the following *enum* constants

   ```
   enum Seasons { Winter, Spring, Summer, Fall };
   ```

   Description
   what is the output of the following code sequence?

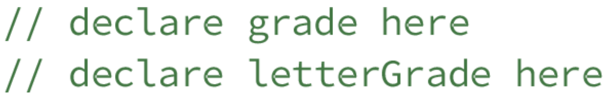
   

   ```
   System.out.println( Seasons.Spring.ordinal( ) );
   ```

Description

7.19.3 Fill In the Code

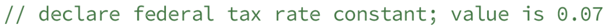
1. Declare two instance variables: *grade*, which is an integer, and *letterGrade*, which is a *char*.



```
// declare grade here
// declare letterGrade here
```

Description

2. Declare a class field for a federal tax rate, a constant, with value .07.

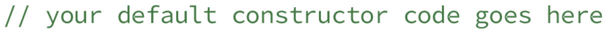


```
// declare federal tax rate constant; value is 0.07
```

Description

For Questions 30 through 37, we will assume that class *TelevisionChannel* has three fields: *name*, a *String*; *number*, an integer; and *cable*, a *boolean*, which represents whether the channel is a cable channel.

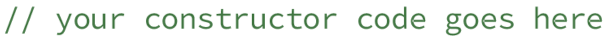
1. Code a default constructor for that class: initialize the fields to an empty string, 0, and *false*, respectively.



```
// your default constructor code goes here
```

Description

2. Code a constructor for that class that takes three parameters.



```
// your constructor code goes here
```

Description

3. Code the three accessors for that class.

```
// your code goes here
```

Description

4. Code the three mutators for that class.

```
// your code goes here
```

Description

5. Code the *toString* method.

```
// your code goes here
```

Description

6. Code the *equals* method.

```
// your code goes here
```

Description

7. Code a method returning the number of digits in the channel number. For instance, if the channel number is 21, the method returns 2; if the channel number is 412, the method returns 3.

```
// your code goes here
```

Description

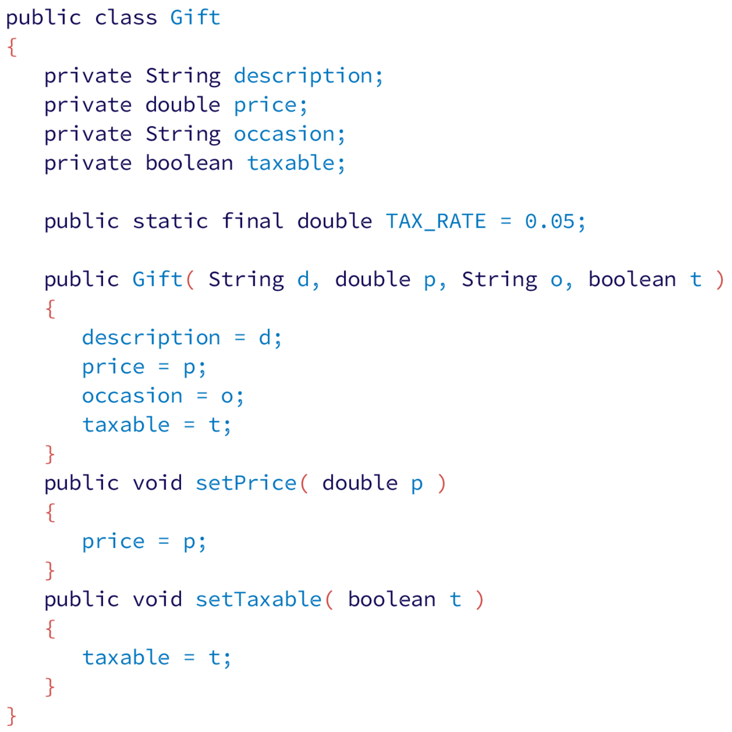
8. Code a method returning the word *cable* if the current object represents a cable channel and returning the word *network* if the current object does not represent a cable channel.

```
// your code goes here
```

Description

7.19.4 Identifying Errors in Code

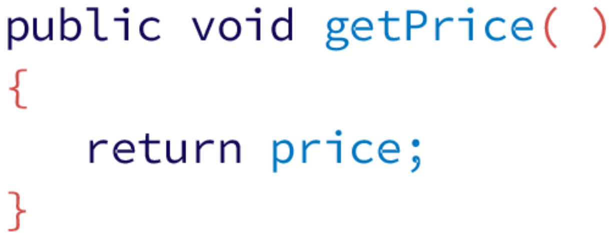
For Questions 38 through 45, consider that inside the class *Gift*, we have already coded the following:



```
public class Gift
{
   private String description;
   private double price;
   private String occasion;
   private boolean taxable;

   public static final double TAX_RATE = 0.05;

   public Gift( String d, double p, String o, boolean t )
   {
      description = d;
      price = p;
      occasion = o;
      taxable = t;
   }
   public void setPrice( double p )
   {
      price = p;
   }
   public void setTaxable( boolean t )
   {
      taxable = t;
   }
}
```

Description

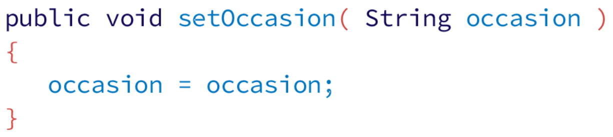
1. We are coding the following inside the class *Gift*; where is the error?



```
public void getPrice( )
{
    return price;
}
```

Description

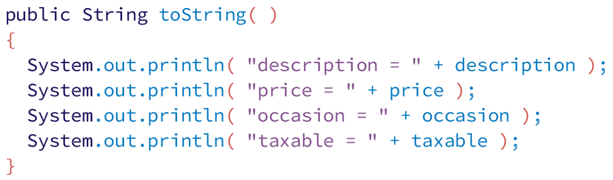
2. We are coding the following inside the class *Gift*; where is the error?



```
public void setOccasion( String occasion )
{
    occasion = occasion;
}
```

Description

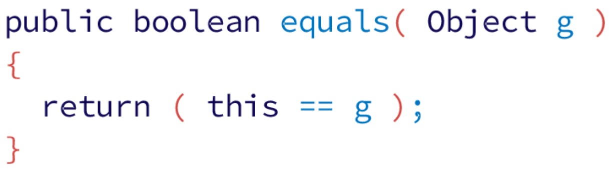
3. We are coding the following inside the class *Gift*; where is the error?



```
public String toString( )
{
  System.out.println( "description = " + description );
  System.out.println( "price = " + price );
  System.out.println( "occasion = " + occasion );
  System.out.println( "taxable = " + taxable );
}
```

Description

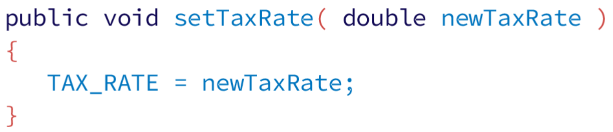
4. We are coding the following inside the class *Gift*; where is the error?



```
public boolean equals( Object g )
{
  return ( this == g );
}
```

Description

5. We are coding the following inside the class *Gift*; where is the error?



```
public void setTaxRate( double newTaxRate )
{
    TAX_RATE = newTaxRate;
}
```

Description

6. We are coding the following inside the class *Gift*; where is

the error?

```
public double calcTax( TAX_RATE )
{
  return ( TAX_RATE * price );
}
```

Description

7. We are coding the following in the *main* method inside the class *GiftClient*; where is the error?

```
Gift g = new Gift( "radio", 59.99, "Birthday", false );
Gift.setPrice( 99.99 );
```

Description

8. We are coding the following in the *main* method inside the class *GiftClient*; where is the error?

```
Gift g = new Gift( "radio", 59.99, "Birthday", false );
g.setTaxable( ) = true;
```

Description

9. Where are the errors in the following statement?

```
enum Months = { "January", "February", "March" };
```

Description

7.19.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

For Questions 47 and 48, consider the following class *Grade*:

```
public class Grade
{
  private char letterGrade;
    public Grade( char lg )
```

Description

```
public class Grade
{
  private char letterGrade;
     public Grade( char lg )
  {
    letterGrade = lg;
  }
  public char getLetterGrade( )
  {
    return  letterGrade;
  }
  public void setLetterGrade( char lg )
  {
    letterGrade = lg;
  }
}
```

1. In the *main* method of the class *GradeClient*, you have coded

   ```
   Grade g = new Grade( 'B' );
   g.letterGrade = 'A';          // line 10
   ```

   Description

   When you compile, you get the following message:

   ```
   GradeClient.java:10: error: letterGrade has private access
   in Grade
   g.letterGrade = 'A';          // line 10
    ^
   1 error
   ```

   Description

   Explain what the problem is and how to fix it.

2. In the *main* method of the class *GradeClient*, you have coded

```
Grade g = new Grade( "A" );   // line 10
```

Description

When you compile, you get the following message:

```
GradeClient.java:10: error: incompatible types: String
cannot be converted to char
Grade g = new Grade ( "A" ); // line 10
                      ^
Note: Some messages have been simplified; recompile with
-Xdiags:verbose to get full output
1 error
```

Description

Explain what the problem is and how to fix it.

3. You coded the following definition for the class *Grade*:

```
public class Grade
{
    private char letterGrade;
    public char Grade( char startLetter )
    {
       letterGrade = startLetter;
    } // line 8
}
```

Description

When you compile, you get the following message:

```
Grade.java:8: error: missing return statement
 } // line 8
 ^
1 error
```

Description

Explain what the problem is and how to fix it.

4. You coded the following definition for the class *Grade*:

```
public class Grade
{
  private char letterGrade;

  public Grade( char lg )
  {
    letterGrade = lg;
  }
  public String toString( )     // line 10
  {                             // line 11
     return letterGrade;        // line 12
  }                             // line 13
}
```

Description

When you compile, you get the following message:

```
Grade.java:12: error: incompatible types: char cannot be
converted to String
return letterGrade; // line 12
       ^
1 error
```

Description

Explain what the problem is and how to fix it.

5. You coded the following definition for the *Grade* class:

```
public class Grade
{
  private char letterGrade;
  public Grade( char lg )
  {
    letterGrade = lg;
  }
  public String toString( )  // line 10
  {                          // line 11
    return lg;               // line 12
  }                          // line 13
}
```

Description

When you compile, you get the following message:

```
Grade.java:12: error: cannot find symbol
 return lg; // line 12
        ^
   symbol  : variable lg
   location: class Grade
1 error
```

Description

Explain what the problem is and how to fix it.

6. You coded the following definition for the *Grade* class:

```
public class Grade
{
  private int numberGrade;
  public Grade( int numberGrade )
  {
    numberGrade = numberGrade;
  }
  public int getGrade( )
  {
    return numberGrade;
  }
}
```

Description

In the *main* method of the *GradeClient* class, you have coded:

```
Grade g1 = new Grade( 95 );
System.out.println( g1.getGrade( ) );
```

Description

The code compiles properly and runs, but the result is not what you expected.

The client’s output is 0, not 95.

Explain what the problem is and how to fix it.

7. You have defined the following *enum* constants:

```
enum Seasons { Winter, Spring, Summer, Fall };
```

Description

In the *main* method of the class *Test*, you have coded:

```
Seasons s = Seasons.Spring;
if ( s.equals( Winter ) )    // line 10
      System.out.println( "It is cold" );
else
      System.out.println( "The weather is fine" );
```

Description

When you compile, you get the following message:

```
Test.java:10: error: cannot find symbol
  if ( s.equals( Winter ) )  // line 10
                  ^
   symbol  : variable Winter
   location: class Test
1 error
```

Description

Explain what the problem is and how to fix it.

8. You have defined the following *enum* constants:

```
enum Seasons { Winter, Spring, Summer, Fall };
```

Description

In the *main* method of the class Test, you have coded

```
Seasons.Fall = Autumn; // line 10
```

Description

When you compile, you get the following message:

```
Test.java:10: error: cannot assign a value to final variable
Fall
  Seasons.Fall = Autumn;  // line 10
         ^
Test.java:10: cannot find symbol
  Seasons.Fall = Autumn;  // line 10
                 ^
   symbol  : variable Autumn
   location: class Test
2 errors
```

Description
Explain what the problem is and how to fix it.

7.19.6 Write a Short Program

1. Write a class encapsulating the concept of a team (for example, “Orioles”), assuming a team has only one attribute: the team name. Include a constructor, the accessor and mutator, and methods *toString* and *equals*. Write a client class to test all the methods in your class.
2. Write a class encapsulating the concept of a television set, assuming a television set has the following attributes: a brand and a price. Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Write a client class to test all the methods in your class.
3. Write a class encapsulating the concept of a course grade, assuming a course grade has the following attributes: a course name and a letter grade. Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Write a client class to test all the methods in your class.
4. Write a class encapsulating the concept of a course, assuming a course has the following attributes: a code (for instance, CS1), a description, and a number of credits (for instance, 3). Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Write a client class to test all the methods in your class.
5. Write a class encapsulating the concept of a student, assuming a student has the following attributes: a name, a Social Security number, and a GPA (for instance, 3.5). Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Write a client class to test all the methods in your class.

6. Write a class encapsulating the concept of website statistics, assuming website statistics have the following attributes: number of visitors and type of site (commercial, government, etc.). Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Write a client class to test all the methods in your class.
7. Write a class encapsulating the concept of a corporate name (for example, “IBM”), assuming a corporate name has only one attribute: the corporate name itself. Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Also include a method returning a potential domain name by adding *www.* at the beginning and *.com* at the end of the corporate name (for instance, if the corporate name is IBM, that method should return *www.ibm.com*). Write a client class to test all the methods in your class.
8. Write a class encapsulating the concept of a file, assuming a file has only a single attribute: the name of the file. Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Also, code a method returning the extension of the file; that is, the letters after the last dot in the file (for instance, if the file name is *Test.java*, then the method should return *java*); if there is no dot in the file name, then the method should return “*unknown extension*.” Write a client class to test all the methods in your class.
9. Writing *System.out.println* every time we want to output a message is annoying. Write a class that enables us to write *S.pln(“Hello world”)* instead. Test your class in a client.
10. Write a class that enables us to write code like:

```
int number = Utility.getInt( "Enter an int", '>' );
```

Description
to replace something like:

```
System.out.println( "Enter an int > " );
Scanner scan = new Scanner( System.in );
int number = scan.nextInt( );
```

Description
Test your class in a client.

7.19.7 Programming Projects

1. Write a class encapsulating the concept of the weather forecast, assuming that it has the following attributes: the temperature and the sky conditions, which could be sunny, snowy, cloudy, or rainy. Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Temperature, in Fahrenheit, should be between –50 and +150; the default value is 70, if needed. The default sky condition is sunny. Include a method that converts Fahrenheit to Celsius. Celsius temperature = (Fahrenheit temperature – 32) * 5 / 9. Also include a method that checks whether the weather attributes are consistent (there are two cases where they are not consistent: when the temperature is above 32 and it is snowy, and when the temperature is below 32 and it is rainy). Write a client class to test all the methods in your class.
2. Write a class encapsulating the concept of a domain name, assuming a domain name has a single attribute: the domain name itself (for instance, *www.yahoo.com*). Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Also include the following methods: one returning whether the domain name starts with *www*; another returning the extension of the domain name (i.e.,

the letters after the last dot, for instance *com*, *gov*, or *edu*; if there is no dot in the domain name, then you should return "*unknown*"); and another returning the name itself (which will be the characters between *www* and the extension; for instance, *yahoo* if the domain is *www.yahoo.com*—if there are fewer than two dots in the domain name, then your method should return "*unknown*"). Write a client class to test all the methods in your class.

3. Write a class encapsulating the concept of an HTML page, assuming an HTML statement has only a single attribute: the HTML code for the page. Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Include the following methods: one checking that there is a > character following each < character, one counting how many images are on the page (i.e., the number of *img* tags), and one counting how many links are on the page (i.e., the number of times we have "*a href*"). Write a client class to test all the methods in your class.
4. Write a class encapsulating the concept of coins, assuming that coins have the following attributes: a number of quarters, a number of dimes, a number of nickels, and a number of pennies. Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Also code the following methods: one returning the total amount of money in dollar notation with two significant digits after the decimal point, and others returning the money in quarters (for instance, 0.75 if there are three quarters), in dimes, in nickels, and in pennies. Write a client class to test all the methods in your class.
5. Write a class encapsulating the concept of a user-defined *double*, assuming a user-defined *double* has only a single

attribute: a *double*. Include a constructor, the accessor and mutator, and methods *toString* and *equals*. Add a method, taking one parameter specifying how many significant digits we want to have, and returning a *double* representing the original *double* truncated so that it includes the specified number of significant digits after the decimal point (for instance, if the original *double* is 6.9872 and the argument of the method is 2, this method will return 6.98). Write a client class to test all the methods in your class.

6. Write a class encapsulating the concept of a circle, assuming a circle has the following attributes: a *Point* representing the center of the circle, and an integer, the radius of the circle. Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Also include methods returning the perimeter (2 * π * radius) and area (π * radius^2) of the circle. Write a client class to test all the methods in your class.
7. Write a class encapsulating the concept of a rational number, assuming a rational number has the following attributes: an integer representing the numerator of the rational number, and another integer representing the denominator of the rational number. Include a constructor, the accessors and mutators, and methods *toString* and *equals*. You should not allow the denominator to be equal to 0; you should give it the default value 1 in case the corresponding argument of the constructor or a method is 0. Also include methods performing multiplication of a rational number by another and addition of a rational number to another, returning the resulting rational number in both cases. Write a client class to test all the methods in your class.

8. Write a class encapsulating the concept of an investment, assuming the investment has the following attributes: the amount of the investment and the interest rate at which the investment will be compounded. Include a constructor, the accessors and mutators, and methods *toString* and *equals*. Also include a method returning the future value of the investment depending on how many years we hold it before selling it, which can be calculated using the following formula:

   $$\text{future value} = \text{investment}\ (\ 1 + \text{interest rate}\ )^{\text{numberOfYears}}$$

   We will assume that the interest rate is compounded annually. Write a client class to test all the methods in your class.

9. Write a class encapsulating the concept of a telephone number, assuming a telephone number has only a single attribute: a *String* representing the telephone number. Include a constructor, the accessor and mutator, and methods *toString* and *equals*. Also include methods returning the area code (the first three digits/characters of the phone number; if there are fewer than three characters in the phone number or if the first three characters are not digits, then this method should return “*unknown area code*”). Write a client class to test all the methods in your class. Look up the *getClass* method in the *Object* class and the *getName* method in the *Class* class and use them in

your client.

10. Write a class that encapsulates the concept of a tennis match score (best of 3 sets). It has the following instance variables: two *ints* for the score in the first set, two *ints* for the score in the second set, two *ints* for the score in the third set, and two *Strings* for the score in the current game. You do not need to keep track of who is serving. Your score should comply with basic tennis rules (a game score for a given player is either LOVE, 15, 30, 40, or AD). A set must be won by 2 (there is no tie breaker in this format). Write the following methods: a default constructor, *toString*, a method to update the score (when a player wins a point), and a method to check if the game is over. The method that updates the score takes one parameter, a *boolean*; if it is *true*, the first player won the point, if it is *false*, the second player won the point. Furthermore, if the game is over, the score should not be changed. The method that checks if the game is over returns *0* if the game is not over,

1 if the first player won, and *2* if the second player won. Write a client that enables the user to play.

11. Write a class that encapsulates an icon for your favorite TV channel or Internet website. Your icon should be made of a minimum of three colors and three shapes. It should be scalable by specifying the value of one instance variable, and the client should be able to change the colors easily. Your class should include a constructor, a draw method, and mutators to enable the user to change the scale, to change the colors in the icon, and to change the top-left coordinates of the icon.

7.19.8 Technical Writing

1. An advantage of object-oriented programming is code reuse, not just by the programmer who wrote the class, but by other programmers. Describe the importance of proper documentation and how you would document a class so that other programmers can use it easily.
2. Java has a number of naming conventions for classes, methods, and field variables. Is this important? Why is it good to respect these conventions?

7.19.9 Group Project (for a group of 1, 2, or 3 students)

1. Write a program that solves a quadratic equation in all cases, including when both roots are complex numbers. For this, you need to set up the following classes:

   Complex, which encapsulates a complex number
   ComplexPair, which encapsulates a pair of complex numbers
   Quadratic, which encapsulates a quadratic equation

SolveEquation, which contains the *main* method

Along with the usual constructors, accessors, and mutators, you will need to code additional methods:

In the *Complex* class, a method that determines whether a complex object is real

In the *ComplexPair* class, a method that determines whether both complex numbers are identical

In the *Quadratic* class, a method to solve the quadratic equation and return a *ComplexPair* object

Additionally, you need to include code in the *main* method to solve several examples of quadratic equations input from the keyboard. Your output should make comments as to what type of roots we get (double real root, distinct real roots, distinct complex roots). You should check that your code works in all four basic cases:

- The quadratic equation is actually a linear equation.
- Both roots are complex.
- There is a double real root.
- There are two distinct real roots.

© Lois GoBe/Shutterstock.

CHAPTER 8

Single-Dimensional Arrays

CHAPTER CONTENTS

Introduction
8.1 Declaring and Instantiating Arrays

- 8.1.1 Declaring Arrays
- 8.1.2 Instantiating Arrays
- 8.1.3 Combining the Declaration and Instantiation of Arrays
- 8.1.4 Assigning Initial Values to Arrays

8.2 Accessing Array Elements
8.3 Aggregate Array Operations

- 8.3.1 Printing Array Elements
- 8.3.2 Reading Data into an Array
- 8.3.3 Summing the Elements of an Array
- 8.3.4 Finding Maximum or Minimum Values
- 8.3.5 Copying Arrays
- 8.3.6 Changing the Size of an Array
- 8.3.7 Comparing Arrays for Equality

    8.3.8 Displaying Array Data as a Bar Chart

8.4 **Programming Activity 1: Working with Arrays**
8.5 Using Arrays in Classes

    8.5.1 Using Arrays in User-Defined Classes
    8.5.2 Retrieving Command Line Arguments

8.6 Searching and Sorting Arrays

    8.6.1 Sequential Search of an Unsorted Array
    8.6.2 Selection Sort
    8.6.3 Insertion Sort
    8.6.4 Sorting Arrays of Objects
    8.6.5 Sequential Search of a Sorted Array
    8.6.6 Binary Search of a Sorted Array

8.7 **Programming Activity 2: Searching and Sorting Arrays**
8.8 Using Arrays as Counters
8.9 Methods Accepting a Variable Number of Arguments
8.10 Chapter Summary
8.11 Exercises, Problems, and Projects

    8.11.1 Multiple Choice Exercises
    8.11.2 Reading and Understanding Code
    8.11.3 Fill In the Code
    8.11.4 Identifying Errors in Code
    8.11.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
    8.11.6 Write a Short Program
    8.11.7 Programming Projects
    8.11.8 Technical Writing
    8.11.9 Group Project

Introduction

Up to this point, we have been working with individual, or scalar, variables; that is, each variable has held one value at a time. To process a group of variables of the same type—for example, counting the number of odd integers entered by the user—we used a *while* loop or a *for* loop.

Thus, to find the average high temperature for the last year, we would use a *for* loop:

```
double dailyTemp;
double total = 0.0;
for ( int i = 1; i <= 365; i++ )
{
   System.out.print( "Enter a temperature" );
   dailyTemp = scan.nextDouble( );
   total += dailyTemp;
}
double average = total / 365;
```

Description

We defined one variable, *dailyTemp*, to hold the data. We read each temperature into our *dailyTemp* variable, added the temperature to our total, then read the next value into the *dailyTemp* variable, added that temperature to the total, and so on, until we finished reading and processing all the temperatures. Each time we read a new temperature, it overwrote the previous temperature, so that at the end of the loop, we had access to the last temperature only.

But suppose we want to perform multiple operations on those temperatures. Perhaps we want to find the highest or lowest temperature or find the median. Or suppose we don’t know what operations we will perform, or in what order, until the user chooses them from a menu. In those cases, one scalar variable, *dailyTemp*, won’t work; we want to store all the temperatures in memory at the same time. An array allows us to do just that without declaring 365 variables individually.

An **array** is a sequence of variables of the same data type. The data type could be any Java primitive data type, such as *int, float, double, byte, boolean, char, short*, or *long*, or it could be a class. Each variable in the array, called an **element**, is accessed using the array name and a subscript, called an **index**, which refers to the element's position in the array.

Arrays are useful for many applications: for example, calculating statistics on a group of data values or processing data stored in tables, such as matrices or game boards.

8.1 Declaring and Instantiating Arrays

In Java, arrays are implemented as objects, so creating an array takes two steps:

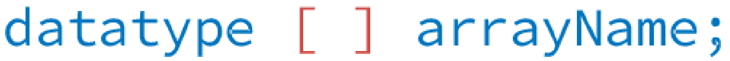
1. Declaring the object reference for the array.
2. Instantiating the array.

In arrays of primitive types, each element in the array contains a value of that type. For example, in an array of *doubles*, each element contains a *double* value. In arrays of objects, each element is an object reference, which stores the location of an object.

8.1.1 Declaring Arrays

To declare an array, we specify the name of the array and the data type, as we would for any other variable. Adding an empty set of brackets ([ ]) indicates that the variable is an array.

Here is the syntax for declaring an array:



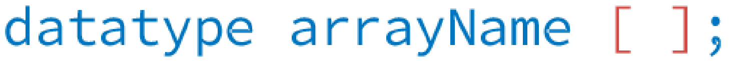
```
datatype [ ] arrayName;
```

Description

For example, the following statement creates a reference to an array that will hold daily high temperatures:

```
double [ ] dailyTemps; // each element is a double
```

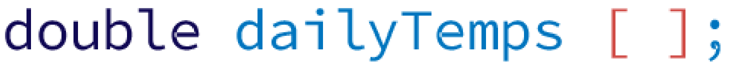
The brackets can be placed before or after the array name. So the following syntax is also valid:



```
datatype arrayName [ ];
```

Description

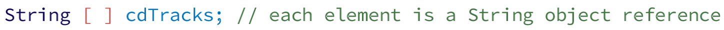
Thus, we could have declared the preceding array using the following statement:



```
double dailyTemps [ ];
```

Although Java code can be written using either syntax, we prefer the first format with the brackets right after the data type, because it’s easier to read as “a *double* array.”

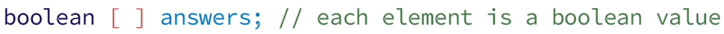
To declare an array to hold the titles of all tracks on a CD, we might declare it this way:



```
String [ ] cdTracks; // each element is a String object reference
```

Description

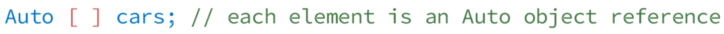
Similarly, this statement declares an array to hold the answers to a true/false test:



```
boolean [ ] answers; // each element is a boolean value
```

Description

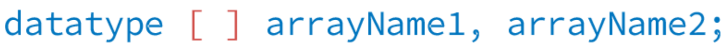
Assuming we have written an *Auto* class, this statement declares an array to hold *Auto* objects:



```
Auto [ ] cars; // each element is an Auto object reference
```

Description

We can declare multiple arrays of the same data type in one statement by inserting a comma after each array name, using this syntax:



```
datatype [ ] arrayName1, arrayName2;
```

Description

For example, the following statement will declare three integer arrays to hold quiz scores for current courses:

```
int [ ] cs101, bio201, hist102; // all elements are int values
```

Description

Note that an array declaration does not specify how many elements the arrays will have. The declaration simply specifies an object reference for the array and the data type of the elements. Thus, **declaring an array does not allocate memory for the array**.

COMMON ERROR TRAP

Putting the size of the array inside the brackets in the array declaration will generate a compiler error.

8.1.2 Instantiating Arrays

As we mentioned earlier, Java arrays are objects, so to allocate memory for an array, we need to instantiate the array using the *new* keyword. Here is the syntax for instantiating an array:

```
arrayName = new datatype [size];
      where size is an expression that evaluates to an integer and
      specifies the number of elements in the array.
```

Description

The following statements will instantiate the arrays declared earlier:

```
dailyTemps = new double [365]; // dailyTemps has 365 elements
cdTracks = new String [15];    // cdTracks has 15 elements
int numberOfQuestions = 30;
answers = new boolean [numberOfQuestions]; // answers has 30 elements
cars = new Auto [3];           // cars has 3 elements
cs101 = new int [5];           // cs101 has 5 elements
bio201 = new int [4];          // bio201 has 4 elements
hist102 = new int [6];         // hist102 has 6 elements
```

Description

When an array is instantiated, the elements are given initial values automatically.

Numeric elements are set to 0, *boolean* elements are set to *false, char* elements are set to the Unicode null character, and object references are set to *null*, as shown in **Table 8.1**.

Default Initial Values of Array Elements

Element Data Type	**Initial Value**
double	0.0
float	0.0
int, long, short, byte	0
char	null character (‘\u0000’)
boolean	*false*
object reference	*null*

Description

Thus, all the elements in the *dailyTemps* array are given an initial value of 0.0; the elements in the *cs101, bio201*, and *hist102* arrays are given an initial value of 0; the elements of the *answers* array are given an initial value of *false*; and the elements of the *cdTracks* and *cars* arrays are given an initial value of *null*.

8.1.3 Combining the Declaration and Instantiation of Arrays

Arrays also can be instantiated when they are declared. To combine the declaration and instantiation of an array, we use this syntax:

```
datatype [ ] arrayName = new datatype [size];
      where size is an expression that evaluates to an integer and
      specifies the number of elements in the array.
```

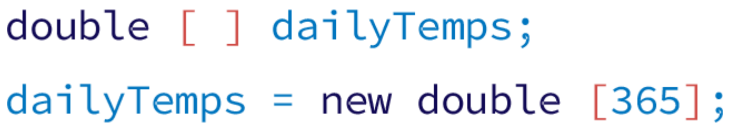
Description

Thus, this statement:

```
double [ ] dailyTemps = new double [365];
```

Description

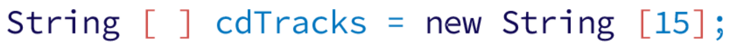
is equivalent to:



```
double [ ] dailyTemps;
dailyTemps = new double [365];
```

Description

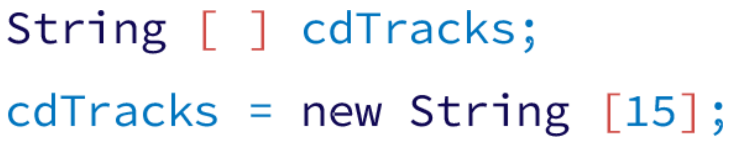
Similarly, this statement:



```
String [ ] cdTracks = new String [15];
```

Description

is equivalent to:

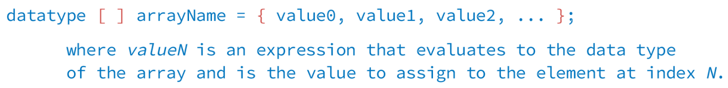


```
String [ ] cdTracks;
cdTracks = new String [15];
```

Description

8.1.4 Assigning Initial Values to Arrays

Java also allows us to instantiate an array by assigning initial values when the array is declared. To do this, we specify the initial values using a comma-separated list within curly braces:



```
datatype [ ] arrayName = { value0, value1, value2, ... };
      where valueN is an expression that evaluates to the data type
      of the array and is the value to assign to the element at index N.
```

Description

Note that we do not use the *new* keyword and we do not specify a size for the array. The number of elements in the array is determined by the number of values in the initialization list.

COMMON ERROR TRAP

An initialization list can be given only when the array is declared. Attempting to assign values to an array using an initialization list after the array is instantiated will generate a compiler error.

For example, this statement declares and instantiates an array of odd numbers:

```
int nine = 9;
int [ ] oddNumbers = { 1, 3, 5, 7, nine, nine + 2, 13, 15, 17, 19 };
```

Description

Because 10 values are given in the initialization list, this array has 10 elements. Notice that the values can be an expression, for example, *nine* and *nine + 2*.

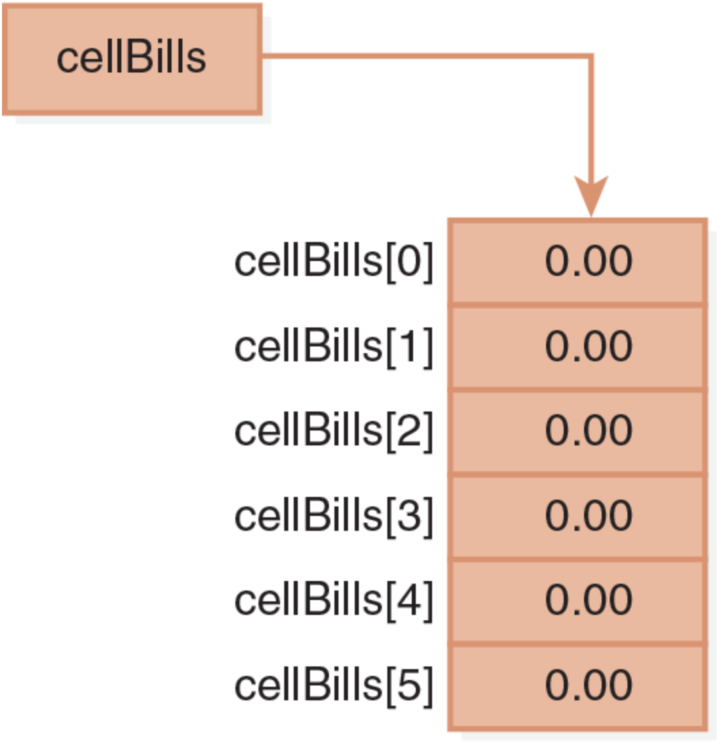
COMMON ERROR TRAP

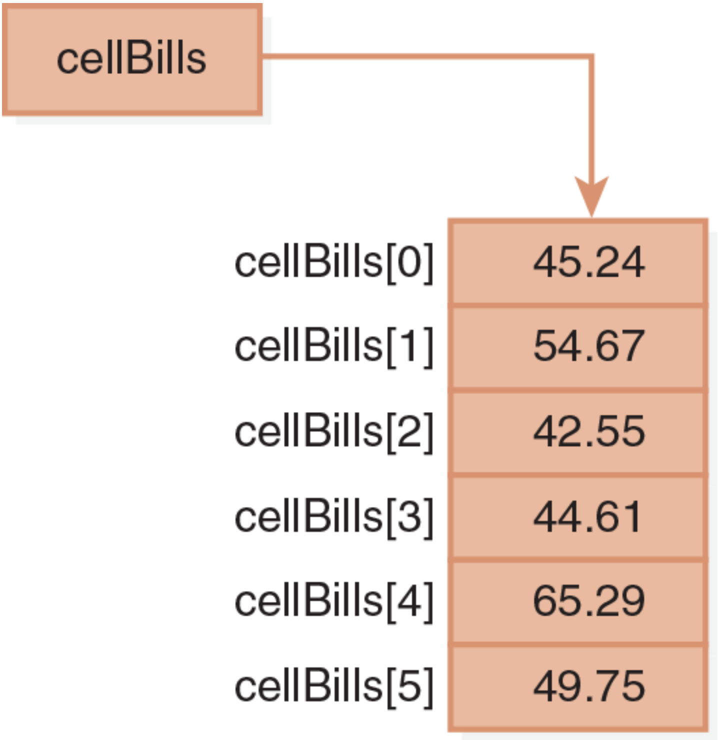
The *new* keyword is not used when an array is instantiated using an initialization list. No size is given; the number of values in the list specifies the size of the array.

Similarly, we can declare and instantiate an array of objects by providing objects in the list, as shown next. The *cars* array of *Auto* objects has three elements.

```
Auto sportsCar = new Auto( "Ferrari", 0, 0.0 );
Auto [ ] cars = { new Auto( "BMW", 100, 15.0 ), sportsCar, new Auto( ) };
```

Description


cellBills
cellBills[0]
cellBills[1]
cellBills[2]
cellBills[3]
cellBills[4]
cellBills[5]
0.00
0.00
0.00
0.00
0.00
0.00


cellBills
cellBills[0]
45.24
cellBills[1]
54.67
cellBills[2]
42.55
cellBills[3]
44.61
cellBills[4]
65.29
cellBills[5]
49.75

```
The first monthly cell bill is 45.24
The last monthly cell bill is 49.75
```

```
cars[0] is a Ferrari
myCar has used 0.0 gallons of gas
Exception in thread "main" java.lang.NullPointerException:
     Cannot invoke "Auto.getModel( )" because "<local1>[2]" is null
     at AutoArray.main(AutoArray.java:26)
```

8.2 Accessing Array Elements

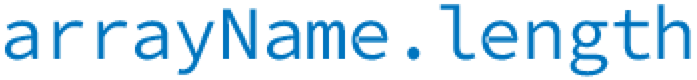
Elements of an array are accessed using this syntax:

```
arrayName[exp]
    where exp is an expression that evaluates to an integer.
```

Description

Exp is the element’s position, or **index**, within the array. The index of the first element in the array is always 0; the index of the last element is always 1 less than the number of elements.

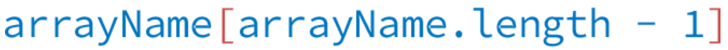
Arrays have a read-only, integer instance variable, **length**, which holds the number of elements in the array. To access the number of elements in an array named *arrayName*, use this syntax:



```
arrayName.length
```

Description

Thus, to access the last element of an array, use this syntax:



```
arrayName[arrayName.length - 1]
```

Description

Note that regardless of the data type of the elements in an array, the *length* of an array is always an integer, because *length* represents the number of elements in the array.

Table 8.2 summarizes the syntax for accessing elements of an array.

Accessing Array Elements

Element	**Syntax**
Element 0	`arrayName[0]`
Element *i*	`arrayName[i]`
Last element	`arrayName[arrayName.length - 1]`

Description

COMMON ERROR TRAP

Note that for an array, *length*—with no parentheses—is an instance variable, whereas for *Strings, length( )*—with parentheses—is a method. Note also that the instance variable is named *length*, rather than *size*.

For example, suppose we want to analyze our monthly cell phone bills for the past six months. We want to calculate the average bill, the total payments for the six months, and the lowest and highest bills. We can use an array of *doubles* with six elements, as shown in **Example 8.1**.

Example 8.1 The *cellBills* Array

```
 1 /* Array of Cell Phone Bills
 2    Anderson, Franceschi
 3 */
 4
 5 public class CellBills
 6 {
 7   public static void main( String [ ] args )
 8   {
 9      // declare and instantiate the array
10      double [ ] cellBills = new double [6];
11
12      // assign values to array elements
13      cellBills[0] = 45.24;
14      cellBills[1] = 54.67;
15      cellBills[2] = 42.55;
16      cellBills[3] = 44.61;
17      cellBills[4] = 65.29;
18      cellBills[5] = 49.75;
19
20      System.out.println( "The first monthly cell bill is "
21                          + cellBills[0] );
22      System.out.println( "The last monthly cell bill is "
23                          + cellBills[cellBills.length - 1] );
24   }
25 }
```

Description

In lines 9–10, we declare and instantiate the *cellBills* array. Because the elements of *cellBills* are *doubles*, instantiating the array also initializes each element to 0.0 and sets the value of *cellBills.length* to 6. Thus, **Figure 8.1** represents the *cellBills* array after line 10 is executed.

Figure 8.1 The *cellBills* Array After Instantiation

Description

Lines 12–18 store values into each element of the array. The element at index *i* of the array is *cellBills[i]*. Remember that the first element of an array is always at index 0. Thus, the last element is *cellBills[5]*, or equivalently, *cellBills[cellBills.length – 1]*. **Figure 8.2** shows how the *cellBills* array looks after lines 13–18 are executed.

Figure 8.2 The *cellBills* Array After Assigning Values

Description

Lines 20–21 print the value of the first element, and lines 22–23 print the value of the last element. The output of Example 8.1 is shown in **Figure 8.3**.

Figure 8.3 Output of Example 8.1

Description

Array indexes *must* be between 0 and *arrayName.length – 1*. Attempting to access an element of an array using an index less than 0 or greater than *arrayName.length – 1* will compile without errors, but will generate an *ArrayIndexOutOfBoundsException* at run time. By default, this exception halts execution of the program.

For example, all the following expressions are invalid:

COMMON ERROR TRAP

Attempting to access an element of an array using an index less than 0 or an index greater than *arrayName.length – 1* will generate an *ArrayIndex-OutOfBounds-Exception* at run time.

```
// invalid indexes for the cellBills array!!
cellBills[-1]                 // the lowest valid index is 0
cellBills[cellBills.length]   // the highest valid index is
                              // cellBills.length - 1
cellBills[150]                // the highest valid index is 5
```

Description

Instantiating an array with a class data type involves two steps:

1. Instantiate the array.

2. Instantiate the objects.

Remember that the elements of an array with a class data type are object references. When the array is instantiated, all elements are set to *null*. Thus, the second step needs to be instantiating each object and assigning its reference to an array element.

Example 8.2 illustrates how to work with an array of objects. In this example, we use an *Auto* class whose API is shown in **Table 8.3**.

Example 8.2 Working with an Array of Objects

```java
/* Working with an Array of Objects
   Anderson, Franceschi
*/

public class AutoArray
{
  public static void main( String [ ] args )
  {
    // 1. instantiate cars array
    Auto [ ] cars = new Auto [3];

    // 2. instantiate Auto objects
    Auto sportsCar = new Auto( "Ferrari", 100, 15.0 );
    cars[0] = sportsCar;     // assign sportsCar to element 0
    cars[1] = new Auto( );   // default Auto object
    // cars[2] has not been instantiated and is null

    // call Auto methods
    System.out.println( "cars[0] is a " + cars[0].getModel( ) );

    Auto myCar = cars[1];
    System.out.println( "myCar has used " + myCar.getGallonsOfGas( )
                         + " gallons of gas" );

    // attempt to call method when Auto object is not instantiated
    System.out.println( "cars[2] is a " + cars[2].getModel( ) );
  }
}
```

Description

The *Auto* Class API

Auto Class Constructor Summary

`Auto( )`

creates an *Auto* object with initial default values of “unknown,” 0, and 0.0.

`Auto( String model, int milesDriven, double gallonsOfGas )`

creates an *Auto* object with the initial values of *model, milesDriven,* and *gallonsOfGas*

Auto Class Method Summary

Return value	**Method name and argument list**
`String`	`getModel( )` returns the value of *model*
`int`	`getMilesDriven( )` returns the value of *milesDriven*
`double`	`getGallonsOfGas( )` returns the value of *gallonsOfGas*
`Auto`	`setModel( String model )` mutator for *model*
`Auto`	`setMilesDriven( int milesDriven )` mutator for *milesDriven*
`Auto`	`setGallonsOfGas( double gallonsOfGas )` mutator for *gallonsOfGas*
`double`	`milesPerGallon( )` returns the gas mileage for this *Auto* object
`String`	`toString( )` returns a *String* representation of this object
`boolean`	`equals( Object obj )` compares this *Auto* object to another object

Description

At lines 9–10, we declare and instantiate *cars*, an array of three *Auto* objects. At this point, each element has the value of *null*. Thus, our second step is to instantiate objects of the *Auto* class and assign their references to the array elements.

At lines 13–14, we instantiate the *Auto* object *sportsCar* and assign the *sportsCar* reference to element 0. At line 15, we instantiate a default *Auto* object and assign its reference to element 1. We do not instantiate an object for element 2, which remains *null*.

COMMON ERROR TRAP

With an array of objects, be sure that an array element points to an instantiated object before attempting to use that element to call a method of the class. Otherwise, a *NullPointer-Exception* will be generated.

We then call methods of the *Auto* class. Because the array elements are object references, to call a method for an object in an array, we use the array name and index, along with the dot notation. This is illustrated in line 19, where we print the model of element 0 by calling the *getModel* method. In lines 21–23, we assign element 1 to the *Auto* reference *myCar*, then call the *getGallonsOfGas* method using the *myCar* reference.

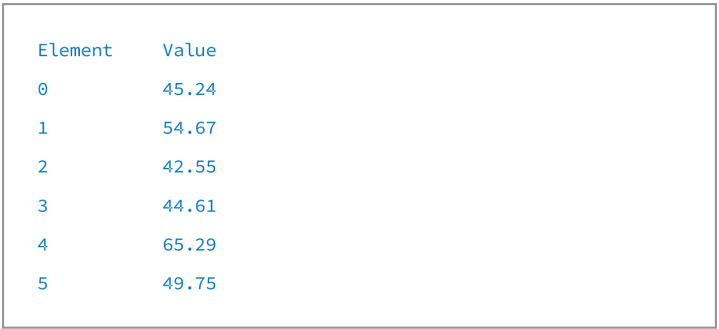
Finally, line 26 attempts to retrieve the model of element 2; however, because *cars[2]* is *null*, a *NullPointerException* is generated. **Figure 8.4** shows the output of this program.

Figure 8.4 Output of Example 8.2

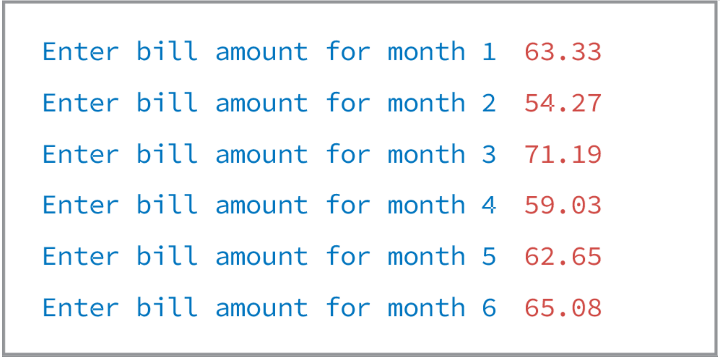
Description

with these end-of-chapter questions

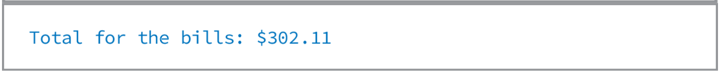
8.11.1	Multiple Choice Exercises Questions 1, 2, 3, 4, 5, 7, 8
8.11.2	Reading and Understanding Code Questions 13, 14, 15
8.11.4	Identifying Errors in Code Questions 36, 37, 38, 40, 42, 44
8.11.5	Debugging Area Question 45



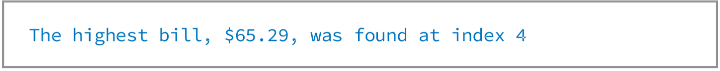
```
Element      Value
0            45.24
1            54.67
2            42.55
3            44.61
4            65.29
5            49.75
```



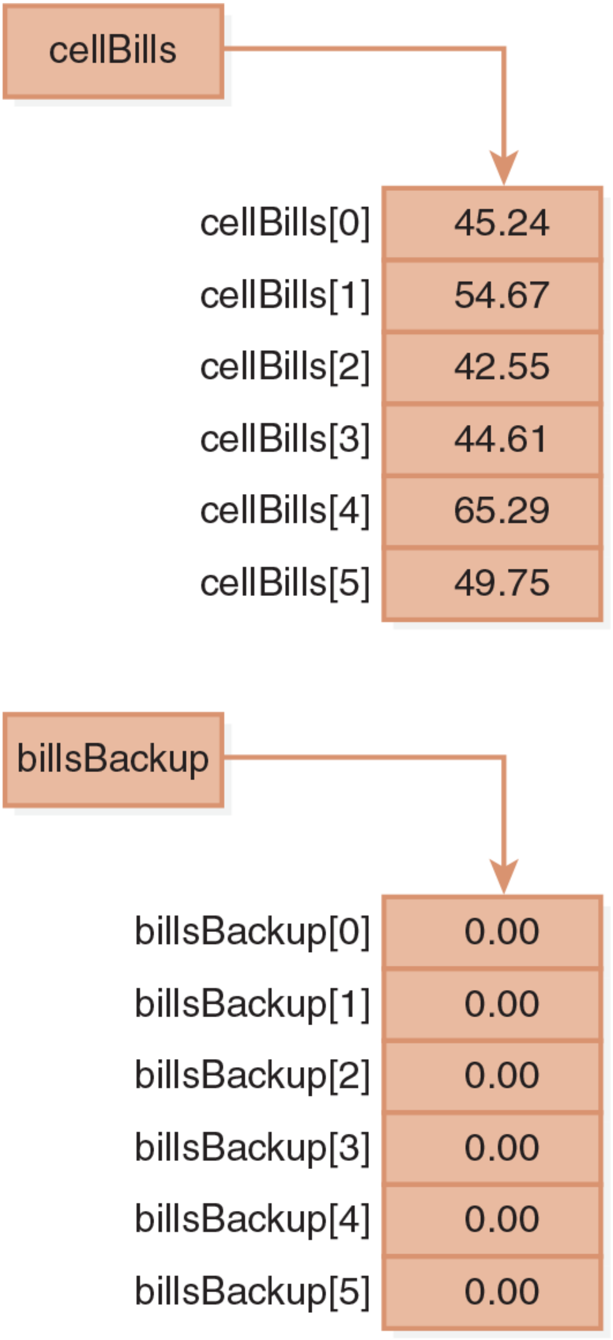
```
Enter bill amount for month 1  63.33
Enter bill amount for month 2  54.27
Enter bill amount for month 3  71.19
Enter bill amount for month 4  59.03
Enter bill amount for month 5  62.65
Enter bill amount for month 6  65.08
```

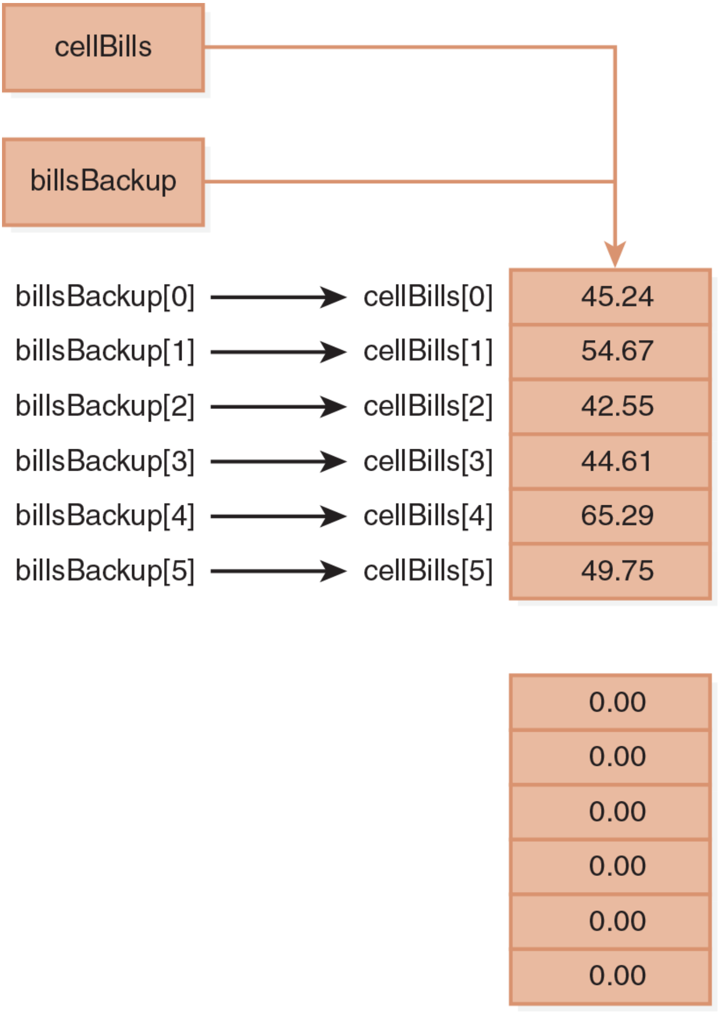


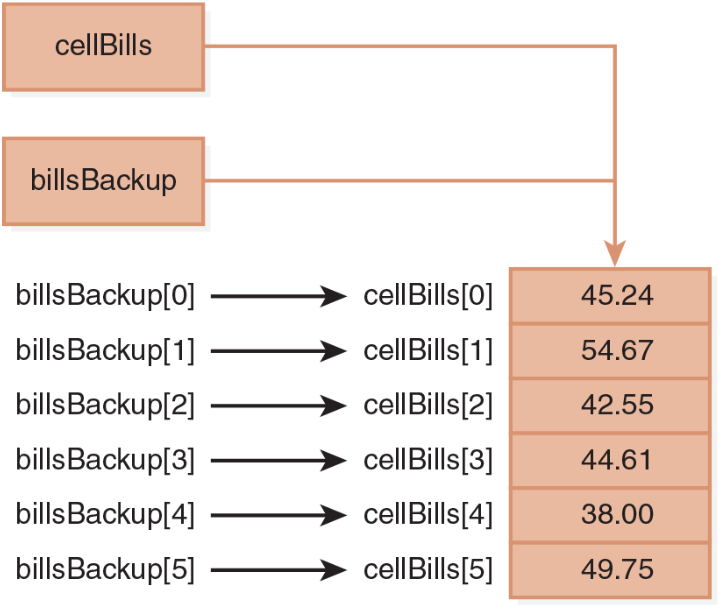
```
Total for the bills: $302.11
```

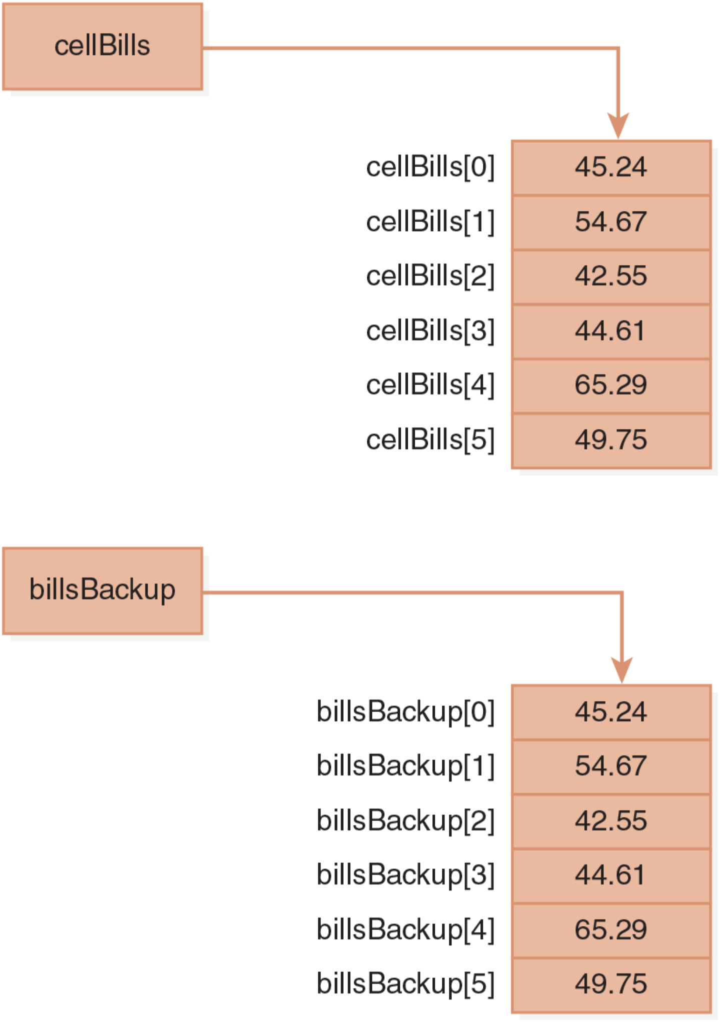


```
The highest bill, $65.29, was found at index 4
```


cellBills
cellBills[0]
45.24
cellBills[1]
54.67
cellBills[2]
42.55
cellBills[3]
44.61
cellBills[4]
65.29
cellBills[5]
49.75
billsBackup
billsBackup[0]
0.00
billsBackup[1]
0.00
billsBackup[2]
0.00
billsBackup[3]
0.00
billsBackup[4]
0.00
billsBackup[5]
0.00

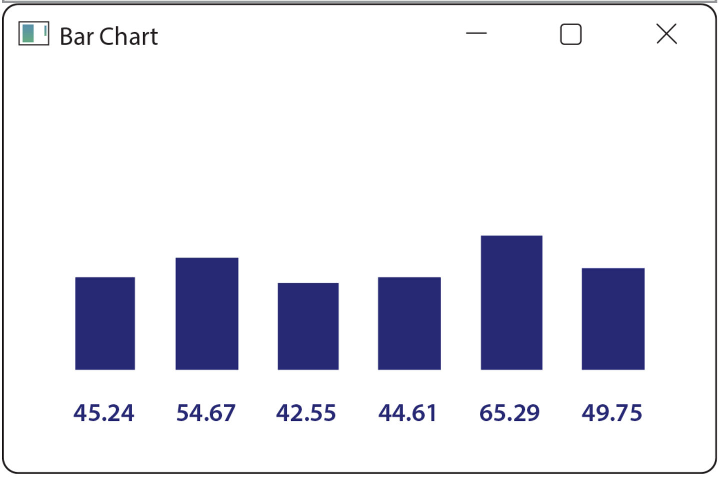

cellBills
billsBackup
billsBackup[0]
cellBills[0]
45.24
billsBackup[1]
cellBills[1]
54.67
billsBackup[2]
cellBills[2]
42.55
billsBackup[3]
cellBills[3]
44.61
billsBackup[4]
cellBills[4]
65.29
billsBackup[5]
cellBills[5]
49.75
0.00
0.00
0.00
0.00
0.00
0.00

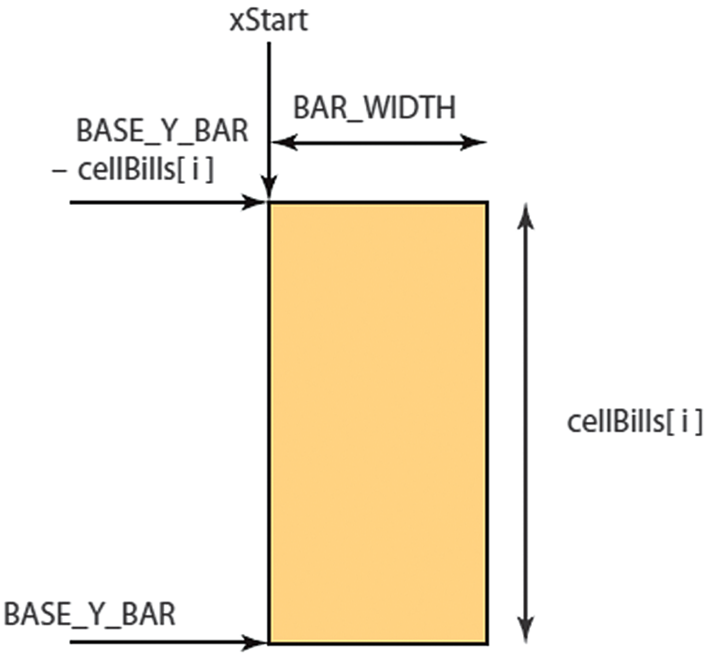

cellBills
billsBackup
billsBackup[0]
cellBills[0]
45.24
billsBackup[1]
cellBills[1]
54.67
billsBackup[2]
cellBills[2]
42.55
billsBackup[3]
cellBills[3]
44.61
billsBackup[4]
cellBills[4]
38.00
billsBackup[5]
cellBills[5]
49.75

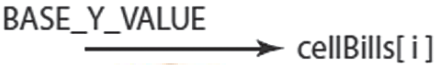

cellBills
cellBills[0] 45.24
cellBills[1] 54.67
cellBills[2] 42.55
cellBills[3] 44.61
cellBills[4] 65.29
cellBills[5] 49.75
billsBackup
billsBackup[0] 45.24
billsBackup[1] 54.67
billsBackup[2] 42.55
billsBackup[3] 44.61
billsBackup[4] 65.29
billsBackup[5] 49.75

```
cellBills
Element       Value
0             45.24
1             54.67
2             42.55
3             44.61
4             65.29
5             49.75

billsBackup
Element       Value
0             45.24
1             54.67
2             42.55
3             44.61
4             38.0
5             49.75
```

```
cellBills1 and cellBills2 are not equal
```


Bar Chart
45.24
54.67
42.55
44.61
65.29
49.75

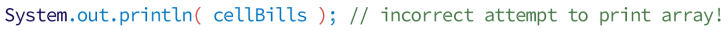

xStart
BAR_WIDTH
BASE_Y_BAR
– cellBills[i]
cellBills[i]
BASE_Y_BAR


BASE_Y_VALUE
cellBills[i]

8.3 Aggregate Array Operations

Once the array is declared and instantiated, it would be convenient if we could just use the array name to perform operations on the whole array, such as printing the array, copying the array to another array, inputting values to the array, and so on. Unfortunately, Java does not support these aggregate operations on arrays.

For example, attempting to print the array using the array name will *not* print all the elements of the array. Instead, this statement:



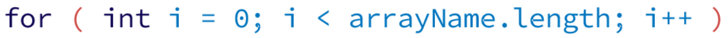
```
System.out.println( cellBills ); // incorrect attempt to print array!
```

Description

calls the *toString* method of the *Array* class, which simply prints the name of the object's class and the hash code of the array name, for example, `[D@310d42`.

8.3.1 Printing Array Elements

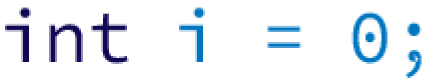
To print all elements of an array, we need to use a loop that prints each element individually. A *for* loop is custom made for processing all elements of an array in order. In fact, the following *for* loop header is a standard way to process all array elements with the loop control variable representing each index in the array:



```
for ( int i = 0; i < arrayName.length; i++ )
```

Description

Note that the initialization statement:

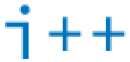


```
int i = 0;
```

Description

sets *i* to the index of the first element of the array.

The loop update

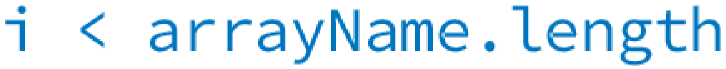


```
i++
```

Description

increments *i* to the next index so that we process each element in order.

The loop condition:



```
i < arrayName.length
```

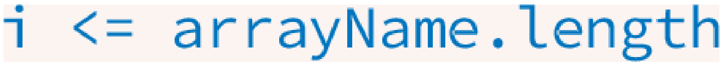
Description

continues execution of the loop as long as the index is less than the *length* of the array.

Note that we use the *less than* operator (<) in the condition. Using the *less than or equal to* operator (<=) would cause us to attempt to reference an element with an index of *arrayName.length*, which is beyond the end of the array.

COMMON ERROR TRAP

In a *for* loop, using the condition:

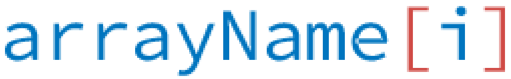


```
i <= arrayName.length
```

Description

will generate an *ArrayIndex-OutOfBounds-Exception* because the index of the last element of an array is *arrayName.length – 1.*

Inside the *for* loop, we refer to the current element being processed as



```
arrayName[i]
```

Description

Example 8.3, whose output is shown in **Figure 8.5**, demonstrates how to print each element in an array.

Example 8.3 Printing All Elements of an Array

```
 1 /* Printing Array Elements
 2    Anderson, Franceschi
 3 */
 4
 5 public class PrintingArrayElements
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     double [ ] cellBills = new double [6];
10     cellBills[0] = 45.24;
11     cellBills[1] = 54.67;
12     cellBills[2] = 42.55;
13     cellBills[3] = 44.61;
14     cellBills[4] = 65.29;
15     cellBills[5] = 49.75;
16
17     System.out.println( "Element\tValue" );
18     for ( int i = 0; i < cellBills.length; i++ )
19     {
20       System.out.println( i + "\t" + cellBills[i] );
21     }
22   }
23 }
```

Description

Figure 8.5 Output of Example 8.3

Description

In lines 9–15, we instantiate the *cellBills* array and assign values to its six elements. In line 18, we use the standard *for* loop header. Inside the *for* loop (line 20), we print each element’s index and value.

8.3.2 Reading Data into an Array

Similarly, we can use the standard *for* loop to input data into an array. In **Example 8.4**, we use a *for* loop to prompt the user for each monthly cell phone bill and to assign the input value to the appropriate array elements.

Example 8.4 Reading Data from the Keyboard into an

Array

```
 1 /* Reading data into an array
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6
 7 public class ReadingDataIntoAnArray
 8 {
 9   public static void main( String [ ] args )
10   {
11     Scanner scan = new Scanner( System.in );
12
13     double [ ] cellBills = new double[6];
14     for ( int i = 0; i < cellBills.length; i++ )
15     {
16       System.out.print( "Enter bill amount for month "
17                         + ( i + 1 ) + "\t" );
18       cellBills[i] = scan.nextDouble( ); // read current bill
19     }
20   }
21 }
```

Description

At lines 14–19, our *for* loop prompts the user for a value for each element in the *cellBills* array. Note that our prompt uses the expression (*i* + 1) for the month number. Although array indexes start at 0, people start counting at 1. If we used the array index in the prompt, we would ask the user for the bills for months 0 to 5. By adding 1 to the array index, we are able to prompt the user for months 1 through 6, which are the month numbers that the user expects.

SOFTWARE ENGINEERING TIP

Prompt for data in terms the user understands.

The output of Example 8.4 is shown in **Figure 8.6.**

Figure 8.6 Reading Data into an Array

Description

8.3.3 Summing the Elements of an Array

To sum the elements of the array, we again use the standard *for* loop, as shown in **Example 8.5.**

Example 8.5 Summing the Elements of an Array

```
 1 /* Summing Array Elements
 2    Anderson, Franceschi
 3 */
 4
 5 import java.text.NumberFormat;
 6
 7 public class SummingArrayElements
 8 {
 9  public static void main( String [ ] args )
10  {
11    double [ ] cellBills = new double [6];
12    cellBills[0] = 45.24;
13    cellBills[1] = 54.67;
14    cellBills[2] = 42.55;
15    cellBills[3] = 44.61;
16    cellBills[4] = 65.29;
17    cellBills[5] = 49.75;
18
19    double totalBills = 0.0; // initialize total
20    for ( int i = 0; i < cellBills.length; i++ )
21    {
22      totalBills += cellBills[i];
23    }
24
25    NumberFormat priceFormat = NumberFormat.getCurrencyInstance( );
26    System.out.println( "Total for the bills: "
27                         + priceFormat.format( totalBills ) );
28   }
29 }
```

Description

We fill the *cellBills* array with values at lines 12–17. We declare the *double* variable *totalBills* and initialize it to 0.0 at line 19. The *for* loop, at lines 20–23, adds each element of the array to *totalBills*. We use the *NumberFormat* class to format the value of *totalBills* as currency for output (lines 25–27). The output of Example 8.5 is shown in **Figure 8.7**.

Figure 8.7 Calculating the Total of All Elements

Description

8.3.4 Finding Maximum or Minimum Values

Suppose we want to find the month that has the lowest bill. That would require finding a minimum value in the array and noting its index. Similarly, to find a month with the highest bill, we would need to find a maximum value in the array and note its index.

To find a maximum or minimum value in an array, we use a variation of the standard *for* loop. **Example 8.6** finds the highest array value and its array index for our *cellBills* array of monthly cell bills.

Example 8.6 Finding a Maximum Value in an Array

```
 1 /* Finding the maximum array value
 2    Anderson, Franceschi
 3 */
 4
 5 import java.text.NumberFormat;
 6
 7 public class MaxArrayValue
 8 {
 9  public static void main( String [ ] args )
10  {
11    double [ ] cellBills = new double [6];
12    cellBills[0] = 45.24;
13    cellBills[1] = 54.67;
14    cellBills[2] = 42.55;
15    cellBills[3] = 44.61;
16    cellBills[4] = 65.29;
17    cellBills[5] = 49.75;
18
19    int maxIndex = 0; // initialize to index of first element
20    for ( int i = 1; i < cellBills.length; i++ )
21    {
22     if ( cellBills[i] > cellBills[maxIndex] )
23       maxIndex = i; // save index of maximum value
24    }
25
26    NumberFormat priceFormat = NumberFormat.getCurrencyInstance( );
27    System.out.println ( "The highest bill, "
28                         + priceFormat.format( cellBills[maxIndex] )
29                         + ", was found at index " + maxIndex );
30  }
31 }
```

Description

We start by assuming that the first element is a maximum value. So we initialize an integer variable, *maxIndex*, to 0, at line 19. Then, at lines 20–24, starting at element 1, we step through the array, comparing the value of each element with the element at *maxIndex*. Whenever we find a value higher than the current maximum, we assign its index to *maxIndex* (line 23). When the *for* loop completes, *maxIndex* holds the index of the array element with the highest value. We then print both that index and the corresponding array value at lines 26–29. The output is shown in **Figure 8.8**.

Figure 8.8 Output of Example 8.6

Description

What happens if the array has only one value? Will we still get the correct result? The answer is yes, because the single element will be at index 0. We start by assigning 0 to *maxIndex*. Then the *for* loop body will not execute because the condition will evaluate to *false*. So *maxIndex* will not be changed and remains set to 0.

What happens if more than one element holds the highest value? We find the index of the first element only, because our condition requires that the element value must be greater than the current maximum to change *maxIndex*.

8.3.5 Copying Arrays

Suppose we create a second array to hold a copy of our cell phone bills, as shown in the following statement:

```
double [ ] billsBackup = new double [6];
```

Description

At this point, all elements of the *billsBackup* array are initialized automatically to 0.0. **Figure 8.9** shows the current state of the *cellBills* and *billsBackup* arrays.

Figure 8.9 The *cellBills* and *billsBackup* Arrays

Description

Then, if we want to copy the elements of the *cellBills* array to the corresponding elements of the *billsBackup* array, we might be tempted to use the assignment operator:

```
billsBackup = cellBills; // incorrect attempt to copy array elements!
```

Description

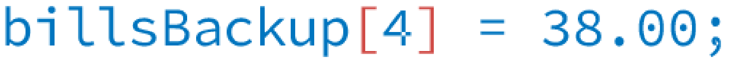
This won't work. Because arrays are objects, the assignment operator copies the *cellBills* object reference to the *billsBackup* object reference. Both *cellBills* and *billsBackup* now point to the same object. The array data was not copied. In fact, we just lost the original *billsBackup* array. With no object reference pointing to it, the array is a candidate for garbage collection, as shown in **Figure 8.10**.

Figure 8.10 Assigning *cellBills* to *billsBackup*

Description

If we were to assign a new value to an element in the *billsBackup* array, we would in fact change the element in the *cellBills* array, because they are now the same array.

This statement:



```
billsBackup[4] = 38.00;
```

Description

has the effect shown in **Figure 8.11**.

Figure 8.11 Altering *billsBackup* Alters *cellBills* Array

Description

Example 8.7 shows how to copy the elements in one array to another array.

Example 8.7 Copying Array Elements into Another Array

```
 1 /* Copying Array Elements to Another Array
 2    Anderson, Franceschi
 3 */
 4
 5 public class CopyingArrayElements
 6 {
 7  public static void main( String [ ] args )
 8  {
 9    double [ ] cellBills = { 45.24, 54.67, 42.55, 44.61, 65.29, 49.75 };
10
11    double billsBackup [ ] = new double [cellBills.length];
12    for ( int i = 0; i < cellBills.length; i++ )
13    {
14      billsBackup[i] = cellBills[i]; // copy each element
15    }
16
17    billsBackup[4] = 38.00; // change value in billsBackup
18
19    System.out.println( "cellBills\nElement\tValue " );
20    for ( int i = 0; i < cellBills.length; i++ )
21    {
22      System.out.println ( i + "\t" + cellBills[i] );
23    }
24
25    System.out.println( "\nbillsBackup\nElement\tValue " );
26    for ( int i = 0; i < billsBackup.length; i++ )
27    {
28      System.out.println ( i + "\t" + billsBackup[i] );
29    }
30  }
31 }
```

Description

At line 9, we instantiate the array *cellBills* using an initialization list. At line 11, we declare and instantiate the array *billsBackup* to have the same size as the original array *cellBills*. At lines 12–15, we use a standard *for* loop to copy one element at a time from the *cellBills* array to the corresponding element in the *billsBackup* array.

Now the *billsBackup* array and the *cellBills* array are separate arrays with their own copies of the element values, as shown in **Figure 8.12**. Changing an element in one array will have no effect on the value of the corresponding element in the other array.

Figure 8.12 Arrays After Copying Each Element

Description

We illustrate this by assigning a new value to an element in the array *billsBackup* (line 17). Finally, we use two *for* loops to print the contents of both arrays. As **Figure 8.13** shows, the value for element 4 is changed only in the array *billsBackup*.

Figure 8.13 Output of Example 8.7

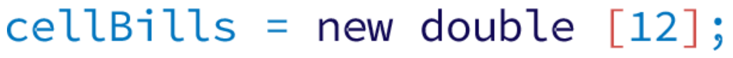
Description

Be aware, however, that when we copy an array whose elements are objects, even using the *for* loop structure, we are copying object references. The result is that the corresponding elements of each array will point to the same object. If an object's data in one array is changed, that change will be reflected in the other array as well.

8.3.6 Changing the Size of an Array

Arrays are assigned a length when they are instantiated, and the *length* of an array becomes a constant value. But what if we want to change the number of elements in an array after it has been instantiated?

For example, our *cellBills* array contains six elements, holding six months' worth of cell phone bills. If we decide to collect a year's worth of cell phone bills, we would need an array with 12 elements. We could instantiate a new version of the *cellBills* array with 12 elements, using this statement:



```
cellBills = new double [12];
```

Description

That statement instantiates a new array of *doubles* all initialized to 0.0. But what happened to the original array of six elements? Since the *cellBills* reference now refers to the new, 12-element array, the 6-element array has no object reference pointing to it, so there is no way we can access the array's values. That is not the result we intended!

To expand the size of an array while maintaining the values of the original array, we can use the following technique:

1. Instantiate an array with the new size, giving the new array

a temporary reference.

2. Copy the elements from the original array to the new array.
3. Point the original array reference to the new array.
4. Assign a *null* value to the temporary array reference.

Thus, instead of immediately pointing *cellBills* to the new array, we should instantiate a 12-element array using a temporary array name, copy the six elements from the *cellBills* array into the 12-element array, assign the *cellBills* reference to the new array, and assign *null* to the temporary array reference. The following code will do that:

```
double [ ] temp = new double [12]; //instantiate new array

// copy all elements from cellBills to temp
for ( int i = 0; i < cellBills.length; i++ )
{
      temp[i] = cellBills[i]; // copy each element
}
cellBills = temp; // assign temp to cellBills
temp = null;      // temp no longer points to cellBills
```

Description

The last statement sets *temp* to *null* so that we don't have two references to the *cellBills* array.

8.3.7 Comparing Arrays for Equality

To compare whether two arrays are equal, first determine if they are equal in length, and then use a *for* loop to compare the corresponding elements in each array. That is, compare element 0 in the first array to element 0 in the second array; compare element 1 in the first array to element 1 in the second array; and so on. If all elements in the first array are equal to the corresponding elements in the second array, then the arrays are equal. **Example 8.8** compares two arrays of *doubles*, a primitive data type.

Example 8.8 Comparing Arrays of Primitive Data Types

```
 1 /* Comparing Arrays of primitive data types
 2    Anderson, Franceschi
 3 */
 4
 5 public class ComparingArrays
 6 {
 7  public static void main( String [ ] args )
 8  {
 9   double [ ] cellBills1 = { 45.24, 54.67, 42.55, 44.61, 65.29, 49.75 };
10   double [ ] cellBills2 = { 45.24, 54.67, 41.99, 44.61, 65.29, 49.75 };
11
12   boolean isEqual = true;
13   if ( cellBills1.length != cellBills2.length )
14   {
15     isEqual = false; // arrays are not the same size
16   }
17   else
18   {
19     for ( int i = 0; i < cellBills1.length && isEqual; i++ )
20     {
21       if ( Math.abs( cellBills1[i] - cellBills2[i] ) > 0.001 )
22       {
23          isEqual = false; // elements are not equal
24       }
25     }
26   }
27
28  if ( isEqual )
29     System.out.println( "cellBills1 and cellBills2 are equal" );
30   else
31     System.out.println( "cellBills1 and cellBills2 are not equal" );
32  }
33 }
```

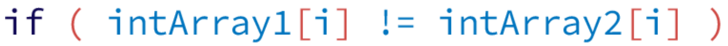
Description

Before we begin the *for* loop, we declare at line 12 a *boolean* variable, *isEqual*, and set it to *true*. In this way, we assume the arrays are equal. Then, our first step is to compare whether the two arrays have the same length (line 13). If they are not the same size, the arrays cannot be equal, so we set *isEqual* to *false* and execution skips to line 28. If the two arrays are the same size, we use a *for* loop at lines 19–25 to test whether the corresponding elements in each array are equal. Note that we have added a second test to the *for* loop condition (*isEqual*). If any corresponding elements are not equal, we set *isEqual* to *false* at line 23. This will cause the condition of the *for* loop to evaluate to *false*, and we exit the *for* loop. Thus, when the *for* loop finishes executing, if any corresponding elements did not match, *isEqual* will be *false*. If both arrays are the same size and all corresponding elements are equal, we never change the value of *isEqual*, so it remains *true*. The output from this example is shown in **Figure 8.14**.

Figure 8.14 Output of Example 8.8

Description

Naturally, if the elements of the arrays are *ints, booleans,* or *chars*, we would use the equality operator (!=) at line 21 as in:

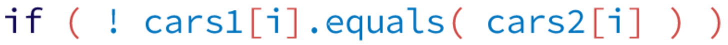


```
if ( intArray1[i] != intArray2[i] )
```

Description

assuming the two arrays we are comparing have names *intArray1* and *intArray2*.

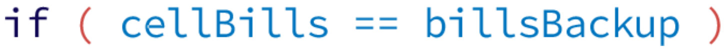
If the elements of the arrays are objects, our *for* loop should call the *equals* method of the objects' class. Thus, to compare two arrays of *Auto* objects, named *cars1* and *cars2*, we would use the following code instead of the condition at line 21:



```
if ( ! cars1[i].equals( cars2[i] ) )
```

Description

A pitfall to avoid is attempting to test whether two arrays are equal using the equality operator (==). This code:



```
if ( cellBills == billsBackup )
```

Description

will not compare the data of the two arrays. It will compare whether the *cellBills* and *billsBackup* object references are equal; that is, whether they point to the same array.

COMMON ERROR TRAP

Because arrays are objects, attempting to compare two arrays using the equality operator (==) will compare whether the two array references point to the same array in memory, not whether the data in the two arrays are equal. Calling the *equals* method inherited from the *Object* class yields similar results.

Similarly, the *equals* method inherited from *Object* also returns the wrong results.

This code:

```
if ( cellBills.equals( billsBackup ) )
```

Description

will return *true* only if both object references point to the same array.

8.3.8 Displaying Array Data as a Bar Chart

One way to display array data is graphically, by drawing a bar chart. For example, the bar chart in **Figure 8.15** displays the data in the *cellBills* array.

Figure 8.15 The *cellBills* Array as a Bar Chart

Description

Each bar is simply a rectangle. **Example 8.9** shows the code to generate Figure 8.15.

Example 8.9 Displaying Array Values as a Bar Chart

```java
 1 /* A bar chart application
 2    Anderson, Franceschi
 3 */
 4
 5 import javafx.application.Application;
 6 import javafx.stage.Stage;
 7 import javafx.scene.canvas.GraphicsContext;
 8 import javafx.scene.paint.Color;
 9
10 public class BarChartApplication extends Application
11 {
12   final int LEFT_MARGIN = 35;         // starting x coordinate
13   final int BASE_Y_BAR  = 150;        // bottom of the bars
14   final int BASE_Y_VALUE = 175;       // bottom of the values
15   final int BAR_WIDTH = 30;           // width of each bar
16   final int SPACE_BETWEEN_BARS = 10;  // pixels between bars
17
18   private double [ ] cellBills
19     = { 45.24, 54.67, 42.55, 44.61, 65.29, 49.75 };
20   private String [ ] months
21       = { "March", "April", "May", "June", "July", "August" };
22
23   @Override
24   public void start( Stage stage )
25   {
26     // set up window title and size
27     GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
28                   stage, "Bar Chart", 350, 200 );
29
30     gc.setFill( Color.BLUE );          // bars will be blue
31     int xStart = LEFT_MARGIN;          // x value for first bar
32
33     for ( int i = 0; i < cellBills.length; i++ )
34     {
35       gc.fillRect( xStart, BASE_Y_BAR - cellBills[i],
36                    BAR_WIDTH, cellBills[i] );
37
38       gc.fillText( Double.toString( cellBills[i] ),
39                    xStart, BASE_Y_VALUE );
40
41       // move to starting x value for next bar
42       xStart += BAR_WIDTH + SPACE_BETWEEN_BARS;
43     }
44   }
45
46   public static void main( String [ ] args )
47   {
48     launch( args );
49   }
50 }
```

Description

To create the bar chart, we use our standard *for* loop at lines 31–41 in the *start* method and call the *fillRect* method of the

GraphicsContext class to draw a rectangle for each element (lines 33–34). We use the *fillText* method at lines 36–37 to display the value of each element.

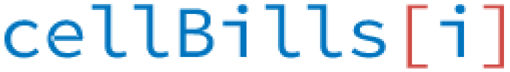
The *fillRect* method takes four arguments: the upper-left *x* value, the upper-left *y* value, the rectangle's width, and the rectangle's height.

We can determine the argument values for the *fillRect* method for each element using the following approach, as illustrated in **Figure 8.16**:

Figure 8.16 Arguments for Drawing Each Bar

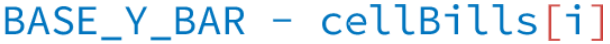
Description

- Width: The width of the bar is a constant value. For our bar chart, we chose a width of 30 pixels; the constant *BAR_WIDTH* stores that value (line 15).
- Height: The height for each bar is the value of the array element being charted. Thus, in the *fillRect* method call (lines 35–36), we represent the height of a bar as:



```
cellBills[i]
```

- Upper-left *y* value: Similarly, the upper-left *y* value will be the height of the bar subtracted from the base *y* value for all the bars; the base *y* value for all the bars is the constant *BASE_Y_BAR* defined in line 13. We subtract the value of the element from the base of the bar because *y* values increase from the top of the window to the bottom. Thus, in our *fillRect* method call, we represent the upper-left *y* value of a bar as:



```
BASE_Y_BAR - cellBills[i]
```

- Upper-left *x* value: We'll start the first bar at the left side of the window, plus a left margin value, represented by the constant *LEFT_MARGIN* (line 12). After we draw each bar, our *for* loop needs to move the starting *x* value to the position of the next bar. To do this, at line 42, we increment the starting *x* value by the width of the bar, *BAR_WIDTH* (defined on line 15), plus the space between bars, *SPACE_BETWEEN_BARS* (defined on line 16).

The arguments to the *fillText* method of the *GraphicsContext* class are the *String* to display and the base *x* and *y* values. At line 36, we convert the *cellBills* element to a *String* using the *toString* method of the *Double* wrapper class. The base *x* value is the same as the starting *x* value for the element's bar, and the base *y* coordinate, *BASE_Y_VALUE*, is the base position for printing the array values (defined on line 14).

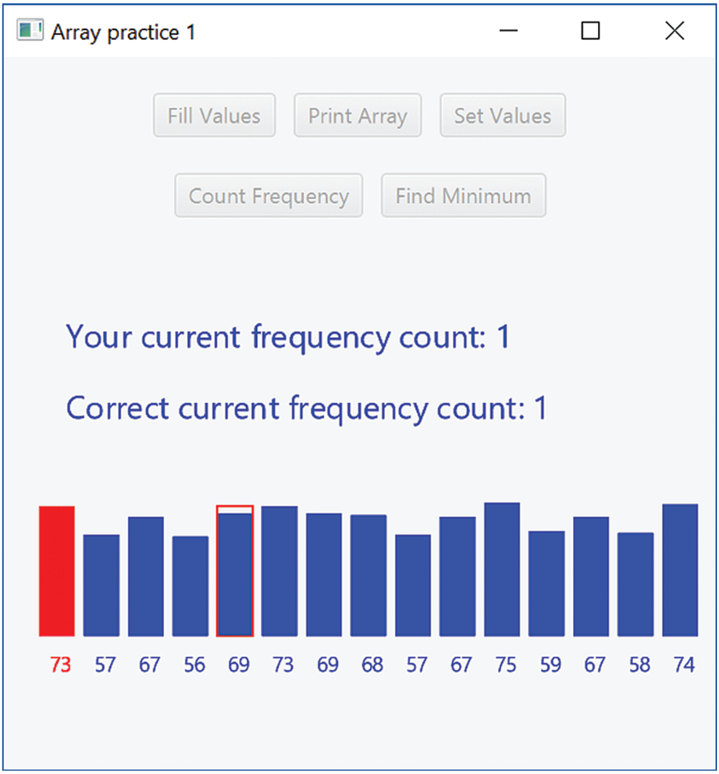
CODE IN ACTION

Within the online resources, you will find a movie with step-by-step illustrations of working with arrays. Click on the link for this chapter to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

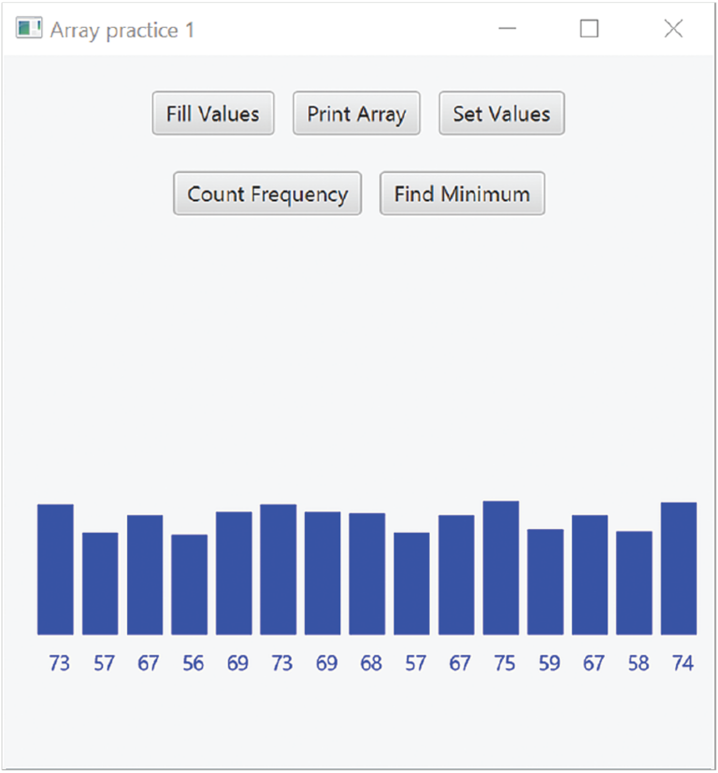
with these end-of-chapter questions

8.11.1	Multiple Choice Exercises Questions 6, 9, 10
8.11.2	Reading and Understanding Code Questions 16, 17, 18, 19, 20, 21
8.11.3	Fill In the Code Questions 27, 28, 29, 30, 31
8.11.4	Identifying Errors in Code Questions 39, 41, 43
8.11.5	Debugging Area Questions 46, 47, 48
8.11.8	Technical Writing Question 76


Array practice 1
Fill Values
Print Array
Set Values
Count Frequency
Find Minimum
73
57
67
56
69
73
69
68
57
67
75
59
67
58
74

Your current frequency count: 1

Correct current frequency count: 1


Array practice 1
Fill Values
Print Array
Set Values
Count Frequency
Find Minimum
73
57
67
56
69
73
69
68
57
67
75
59
67
58
74

8.4 Programming Activity 1: Working with Arrays

In this activity, you will work with a 15-element integer array. Specifically, you will write the code to perform the following operations:

1. fill the array with random numbers between 30 and 99
2. print the array
3. set every array element to a specified value
4. count the number of elements with a specified value
5. find the minimum value in the array

The framework for this Programming Activity will animate your algorithm so that you can check the accuracy of your code.

For example, **Figure 8.17** shows the application counting the elements having the value 73.

Figure 8.17 Animation of the Programming Activity

Description

At this point, the application has found the value 73 in element 0 and is comparing the value 73 with the value 69 in element 4.

Instructions

In this chapter's Programming Activity 1 folder in the supplied code files, you will find the source files needed to complete this activity. Copy all the files to a folder on your computer. Note that all files should be in the same folder.

Open the *ArrayPractice1Controller.java* source file. Searching for five asterisks (*****) in the source code will position you at the sample method and the four other locations where you will add your code. We have provided the sample code for task number 1, which you can use as a model for completing the other tasks. In every task, you will fill in the code for a method that will manipulate an existing array of 15 integers. You should not instantiate the array; we have done that for you. **Example 8.10** shows the section of the *ArrayPractice1Controller* source code where you will add your code.

Example 8.10 Location of Student Code in *ArrayPractice1Controller*

```
// ***** 1. The first method has been coded as an example
/**  Fills the array with random numbers between 50 and 80.
*    The instance variable arr is the integer array
*    to be filled with values
*/
public void fillValues( )
{
     Random rand = new Random( );
      for ( int i = 0; i < arr.length; i++ )
    {
       arr[i] = rand.nextInt( 31 ) + 50;
       animate( i, arr[i] ); // needed to create visual feedback
    }
}
// end of fillValues method

// ***** 2. student writes this method
/** Prints the array to the console with elements separated
*     by a space
*    The instance variable arr is the integer array to be printed
*/
public void printArray( )
{
 // Note: to animate the algorithm, put this method call as the
 // last statement in your for loop:
 //                  animate( i, arr[i] );
 //     where i is the index of the current array element
 // Write your code here:


} // end of printArray method

// ***** 3. student writes this method
/** Sets all the elements in the array to parameter value
*    The instance variable arr is the integer array to be processed
*    @param value the value to which to set the array elements
*/
public void setValues( int value )
{
 // Note: to animate the algorithm, put this method call as the
 // last statement in your for loop
 //            animate( i arr[i] );
 //      where i is the index of the current array element
 // Write your code here:


} // end of setValues method
  // ***** 4. student writes this method
  /** Counts number of elements equal to parameter value
  *    The instance variable arr is the integer array to be processed
  *      @param value the value to count
  *      @return the number of elements equal to value
  */
  public int countFrequency( int value )
  {
   // Note: to animate the algorithm, put this method call as the
   // last statement in your for loop
   //          animate( i, count );
   //         where i is the index of the current array element
   //                 count is the variable holding the frequency
   // Write your code here:


     return 0; // replace this line with your return statement

  } // end of countFrequency method

  // ***** 5. student writes this method
  /** Finds and returns the minimum value in arr
  *    The instance variable arr is the integer array to be processed
  *     @return the minimum value found in arr
  */
  public int findMinimum( )
  {
   // Note: to animate the algorithm, put this method call as the
   // last statement in your for loop
   //        animate( i, minimum );
   //            where i is the index of the current array element
   //               minimum is the variable holding the minimum
   // Write your code here:


     return 0; // replace this line with your return statement

  } // end of findMinimum method

    // End of student code
```

Description

Note that for the *countFrequency* and *findMinimum* methods, we provide a dummy *return* statement (*return 0;*) We do this so that the source code will compile. In this way, you can write and test each method separately, using stepwise refinement. When you are ready to write the *countFrequency* and *findMinimum* methods, just replace the dummy *return* statements with the appropriate *return* statement for that method.

Our framework will animate your algorithm so that you can watch your code work. For this to happen, be sure that your *for* loop calls the *animate* method. The arguments that you send to *animate* will differ depending on the task you are coding. Detailed instructions for each task are included in the code.

To test your code, compile *ArrayPractice1Controller.java* and run the *ArrayPractice1Application* source code. **Figure 8.18** shows the graphics window when the program begins. Because the values of the array are randomly generated, the values will be different each time the program runs. To test any method, click the appropriate button.

Figure 8.18 The Graphics Window When the Application Begins

Description

Troubleshooting

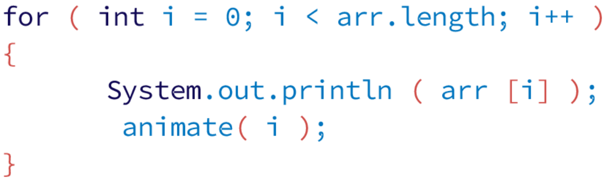
If your method implementation does not animate, follow these tips:

- Verify that the last statement in your *for* loop is a call to the *animate* method and that you passed the appropriate arguments to the *animate* method.
- Verify that your *for* loop has curly braces. For example, the *animate* method call is outside the body of this *for* loop:

```
for ( int i = 0; i< arr.length; i++ )
      System.out.println ( arr [i] );
      animate( i ); // this statement is outside the for loop
```

Description

Remember that without curly braces, the *for* loop body consists of only the first statement following the *for* loop header. Enclosing both statements within curly braces will make the *animate* method call part of the *for* loop body.



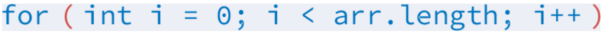
```
for ( int i = 0; i < arr.length; i++ )
{
       System.out.println ( arr [i] );
        animate( i );
}
```

Description

- Verify that you did not instantiate a new array. Perform all operations on the instance variable array named *arr*.

DISCUSSION QUESTIONS

1. Could you use the following *for* loop header in every method? Explain why or why not.



```
for ( int i = 0; i < arr.length; i++ )
```

Description

2. How would you modify the *findMinimum* method to return the index of the minimum value?

```
c1 =
$0.00 $0.00 $0.00 $0.00 $0.00 $0.00


c2 = 555-555-5555
$24.60 $48.75 $62.50


The highest bill is $62.50


The total of all bills is $135.85


c2 and c3 are equal


after client changes original array
c2 = 555-555-5555
$24.60 $48.75 $62.50


after client changes returned array
c2 = 555-555-5555
$24.60 $48.75 $62.50
```

```
The number of parameters is 2
args[0]: input.txt
args[1]: output.txt
```

8.5 Using Arrays in Classes

8.5.1 Using Arrays in User-Defined Classes

An array can be used inside a user-defined class just like any other variable. In particular,

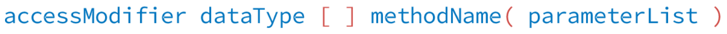
- an array can be an instance variable.
- an array can be a parameter to a method.
- a method can return an array.
- an array can be a local variable inside a method.

To define a method that takes an array as a parameter, use this syntax:

```
accessModifier returnType methodName( dataType [ ] arrayName )
```

Description

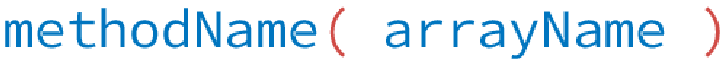
The syntax for a method header that returns an array is



```
accessModifier dataType [ ] methodName( parameterList )
```

Description

To pass an array as an argument to a method, just use the array name without brackets as the argument value:



```
methodName( arrayName )
```

Description

COMMON ERROR TRAP

If we think of the brackets as being part of the data type of the array, then it’s easy to remember that the brackets are included in the method header—where the data types of parameters are given—but that brackets are not included in method calls, where the data itself is given.

In **Example 8.11**, we define a class named *CellPhone* that illustrates the use of arrays in a class.

Example 8.11 The *CellPhone* Class

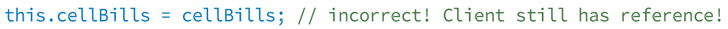
```java
/** CellPhone class
*   Anderson, Franceschi
*/

import java.text.DecimalFormat;

public class CellPhone
{
  public final int MONTHS = 6; // default number of months
  private String phoneNumber;
  private double [ ] cellBills;

  /** Default constructor
  *   creates cellBills with MONTHS elements
  */
  public CellPhone( )
  {
     phoneNumber = "";
     cellBills = new double [MONTHS];
  }

  /** Constructor
  *  @param phoneNumber cell phone number
  *  @param cellBills array of monthly bills
  */
  public CellPhone( String phoneNumber, double [ ] cellBills )
  {
    this.phoneNumber = phoneNumber;

    // instantiate array with same length as parameter
    this.cellBills = new double [cellBills.length];

    // copy parameter array to cellBills array
    for ( int i = 0; i < cellBills.length; i++ )
    {
      this.cellBills[i] = cellBills[i];
    }
  }

  /** Accessor for the phone number
  *   @return the phone number
  */
  public String getPhoneNumber( )
  {
    return phoneNumber;
  }

  /** Accessor for the cell phone bills
  *   @return copy of cellBills array
  */
  public double [ ] getCellBills( )
  {
    double [ ] temp = new double [cellBills.length];
    for ( int i = 0; i < cellBills.length; i ++ )
    {
      temp[i] = cellBills[i];
    }
    return temp;
  }

  /**  Calculates total of all cell phone bills
  *    @return total of all elements in cellBills array
  */
  public double calcTotalBills( )
  {
     double total = 0.0; // initialize total to 0.0

     for ( int i = 0; i < cellBills.length; i++ )
     {
       total += cellBills[i]; // add current element to total
     }
     return total;
  }

  /** Finds a maximum bill
  *  @return largest value in cellBills array
  */
  public double findMaximumBill( )
  {
     double max = cellBills[0]; // assume first element is max

     for ( int i = 1; i < cellBills.length; i++ )
     {
        if ( cellBills[i] > max )
           max = cellBills[i]; // save new maximum
     }
     return max;
  }

   /** Returns printable version of CellPhone object
   *   @return phone number plus each month's bill
   */
   @Override
   public String toString( )
   {
     String returnValue = phoneNumber + "\n";
     DecimalFormat money = new DecimalFormat( "$##0.00" );
    for ( int i = 0; i < cellBills.length; i++ )
    {
        returnValue += money.format( cellBills[i] ) + "\t";
    }
    returnValue += "\n";

    return returnValue;
  }

  /**  Compares two CellPhone objects for equality
  *   @param c CellPhone object
  *   @return true if objects are equal; false, otherwise
  */
  @Override
  public boolean equals( Object c )
  {
    if ( !( c instanceof CellPhone ) )
        return false;
    else
    {
        CellPhone objCP = ( CellPhone ) c;
        if ( !( phoneNumber.equals( objCP.phoneNumber ) ) )
          return false;

        if ( cellBills.length != objCP.cellBills.length )
          return false; // arrays are not the same length

        for ( int i = 0; i < cellBills.length; i++ )
        {
           if ( cellBills[i] != objCP.cellBills[i] )
             return false;
        }
        return true;
    }
  }
}
```

Description

Our *CellPhone* class defines three instance variables in lines 9–11: the phone number (a *String* named *phoneNumber*), monthly bills (an array of *doubles* named *cellBills*), and a constant named *MONTHS*, whose value, 6, represents the number of monthly cell bills, and therefore the length of the *cellBills* array if a *CellPhone* object is instantiated using the default constructor. Note that since *MONTHS* is a constant, we made it *public*.

When our class has instance variables that are arrays, we need to take a little extra care to ensure that encapsulation is not violated.

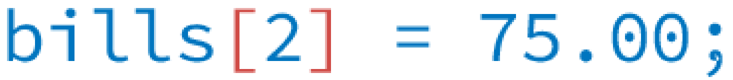
Let's start with initialization of the array. The overloaded constructor of the *CellPhone* class, whose method header is at line 26, includes an array parameter. With parameters of primitive types, the constructor can simply assign the value of the parameter to the instance variable. As we have seen, however, the name of an array is an object reference, which contains the location of the array in memory. If the constructor merely assigns the array parameter, *cellBills*, to our array instance variable, *cellBills*, as in the following code:



```
this.cellBills = cellBills; // incorrect! Client still has reference!
```

Description

then the parameter *cellBills* and the instance variable *cellBills* would point to the same array. That means that the client still has a reference to the array, and the client can change the array values without going through the mutator methods of the class. For example, if the client executes this statement:



```
bills[2] = 75.00;
```

Description

then *cellBills[2]* also gets the value 75.00, because they are the same array. This is clearly a violation of encapsulation, which means that a client should be able to change the *private* fields of a class only by calling the mutator methods of the class.

SOFTWARE ENGINEERING TIP

Sharing array references with the client violates encapsulation. To return an array from a method, copy the elements of the instance variable array to a temporary array and return a reference to the temporary array. Similarly, to accept an array as a parameter to a method, instantiate a new array and copy the elements of the parameter array to the new array.

To avoid this problem, our constructor instantiates a new *cellBills* array that is the same size as the array passed as a parameter, and then copies the elements of the parameter array into the new *cellBills* array (lines 30–37).

There are similar considerations in implementing the accessor method of an array instance variable. With instance variables of primitive types, the accessor methods simply return the value of the instance variable. Our accessor for *cellBills* (lines 48–59) has an array as a return value. If we return the *cellBills* reference, however, we run into the same problem with encapsulation; that is, if our accessor for the *cellBills* instance variable uses this statement:

```
return cellBills; // incorrect! Client has reference to instance variable
```

Description

we give the client a reference to the *cellBills* array, and the client can directly change the values of the array without calling the mutator methods of the class. Just as the constructor instantiated a new array and copied the parameter array's value to the new array, the accessor method should instantiate a new array, copy the *cellBills* array to it, and return a reference to the new array. Thus, at line 53, we declare and instantiate a local array variable named *temp*. At lines 54–57, we copy the contents of *cellBills* into *temp*, and return *temp* at line 58.

We also provide a method *calcTotalBills* (lines 61–73) that calculates the total of the monthly bills using the accumulation technique discussed earlier in the chapter and a *findMaximumBill* method (lines 75–88), which finds a maximum value in the *cellBills* array, also using techniques discussed earlier in the chapter.

Our *toString* method (lines 90–105) builds up a *String* named *returnValue* by first including *phoneNumber*, then formatting each bill using a *DecimalFormat* pattern for money and concatenating that value, plus a tab, to *returnValue*.

The *equals* method (lines 107–132) compares the phone number and each element of the *cellBills* array in the object with the phone number and corresponding element in the *cellBills* array in the parameter object.

We can test our *CellPhone* class with the client class shown in **Example 8.12**. The output is shown in **Figure 8.19**.

Example 8.12 The *CellPhoneClient* Class

```java
/**  Client to exercise the CellPhone class
*    Anderson, Franceschi
*/

import java.text.DecimalFormat;

public class CellPhoneClient
{
   public static void main( String [ ] args )
   {
     double [ ] bills = new double[3]; // array of cell phone bills
     bills[0] = 24.60; // assign values
     bills[1] = 48.75;
     bills[2] = 62.50;

     // instantiate CellPhone object using default constructor
     CellPhone c1 = new CellPhone( );

     // instantiate two identical CellPhone objects
     CellPhone c2 = new CellPhone( "555-555-5555", bills );
     CellPhone c3 = new CellPhone( "555-555-5555", bills );

     // print data from c1 and c2
     System.out.println( "c1 = " + c1.toString( ) );
     System.out.println( "c2 = " + c2.toString( ) );

     // find and print maximum bill
     DecimalFormat money = new DecimalFormat( "$##0.00" );
     System.out.println( "\nThe highest bill is "
                    + money.format( c2.findMaximumBill( ) ) );

     // find and print total of all bills
     System.out.println( "\nThe total of all bills is "
                    + money.format( c2.calcTotalBills( ) ) );

     System.out.println( ); // print blank line
     // call equals method
     if ( c2.equals( c3 ) )
         System.out.println( "c2 and c3 are equal" );
     else
         System.out.println( "c2 and c3 are not equal" );

     // test encapsulation
     // set new value in original array
     bills[2] = 100.00;
     // print c2 to show value in object not changed
     System.out.println( "\nafter client changes original array\n"
                           + "c2 = " + c2.toString( ) );

     // test encapsulation further
     // get array of cell bills and store in new array
     double [ ] billsCopy = c2.getCellBills( );

     billsCopy[1] = 50.00; // change value of one element
     // print c2 to show value in object not changed
     System.out.println( "\nafter client changes returned array\n"
                        + "c2 = " + c2.toString( ) );
  }
}
```

Description

Figure 8.19 Output from the *CellPhoneClient* Class

Description

In the *CellPhoneClient*, we instantiate three *CellPhone* objects. We instantiate *c1* using the default constructor (line 17), giving it an empty phone number and six months of bills initialized to 0.00, as shown in line 24, when we use the *toString* method to print *c1's* data. We set up a *bills* array with three values (lines 11–14) and pass *bills* to the overloaded constructor (lines 20–21) to instantiate *c2* and *c3* with identical data. We then use *toString* to print *c2's* data (line 25).

We then call the *findMaximumBill* method and print its return value (lines 27–30). Next, we call the *calcTotalBills* method and print its return value (lines 32–34).

A call to the *equals* method to compare *c2* and *c3* (lines 37–41) returns a value of *true*, because the two objects have the same data.

Finally, we test encapsulation two ways. First, we change a value in the *bills* array, then print *c2* again to verify that its data has not changed (lines 43–48). Second, we call the accessor method for the *cellBills* array and change a value in the array returned from the method call. We again print *c2* to verify that its data is unchanged (lines 50–57). Testing the *CellPhone* class with such an example is helpful in checking that we have correctly implemented the class.

8.5.2 Retrieving Command Line Arguments

The syntax of an array parameter for a method might look familiar to you. We've seen it repeatedly in Java applications in the header for the *main* method:

```
public static void main( String [ ] args )
```

Description

As we can see, *main* receives a *String* array as a parameter. That array of *Strings* holds the arguments, if any, that the user sends to the program from the command line. An argument might be the name of a file for the program to read or some configuration parameters that specify preferences in how the application should perform its function.

The sample program in **Example 8.13** demonstrates how to retrieve the parameters sent to a Java application. Because *args* is a *String* array, we can use the *length* field to get the number of parameters (lines 8–9), and we use our standard *for* loop format (lines 10–13) to retrieve and print each parameter, as shown in **Figure 8.20**.

Example 8.13 Retrieving Command Line Arguments

```
 1 /** Print Command Line arguments
 2 *   Anderson, Franceschi
 3 */
 4 public class CommandLineArguments
 5 {
 6   public static void main( String [ ] args )
 7   {
 8     System.out.println( "The number of parameters is "
 9                              + args.length );
10     for ( int i = 0; i < args.length; i ++ )
11     {
12         System.out.println( "args[" + i + "]: " + args[i] );
13     }
14   }
15 }
```

Description

Figure 8.20 Output from Example 8.13

Description

Figure 8.20 shows the output produced when we invoke the program as

```
java CommandLineArguments input.txt output.txt
```

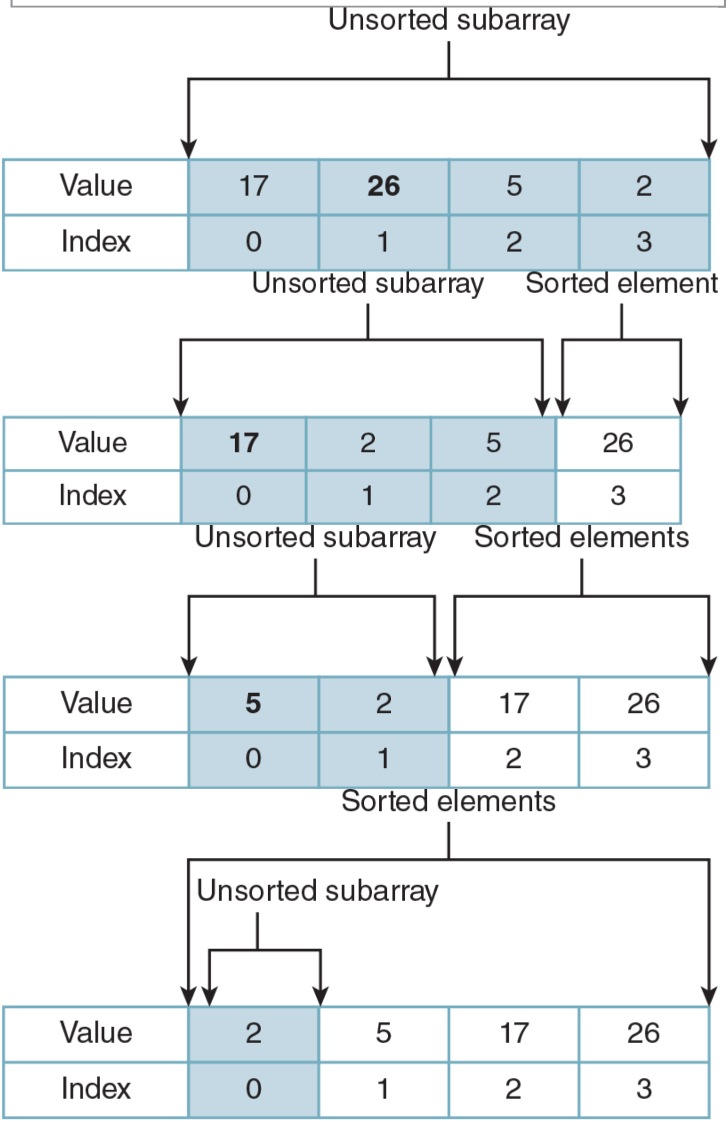
Description

with these end-of-chapter questions

8.11.1	Multiple Choice Exercises Question 11
8.11.2	Reading and Understanding Code

	Questions 22, 23, 24, 25, 26
8.11.3	Fill In the Code Questions 32, 33, 34, 35
8.11.6	Write a Short Program Questions 49, 50, 51, 52, 53, 54, 55, 56, 58, 59, 60, 61

```
Enter the member's ID or 0 to stop > 1234
Sorry, member is not a winner.

Enter the next member's ID or 0 to stop > 3980
You win free popcorn!

Enter the next member's ID or 0 to stop > 0

The winners were 619 4510 3980 4004
```

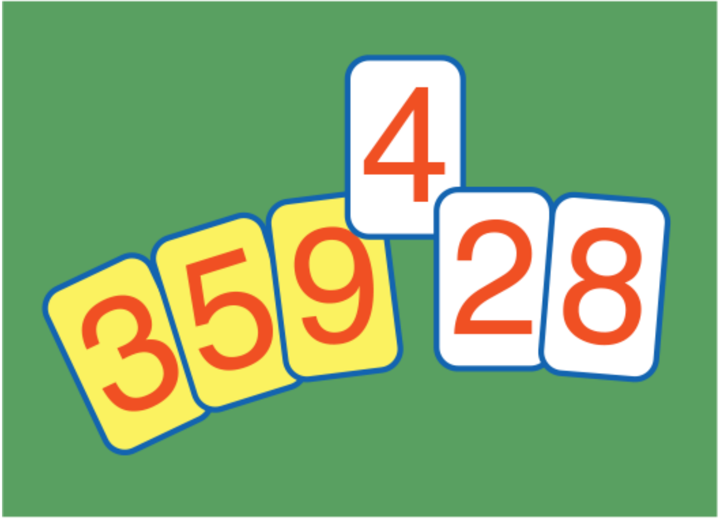

Unsorted subarray
Value
17
26
5
2
Index
0
1
2
3
Unsorted subarray
Sorted element
Value
17
2
5
26
Index
0
1
2
3
Unsorted subarray
Sorted elements
Value
5
2
17
26
Index
0
1
2
3
Sorted elements
Unsorted subarray
Value
2
5
17
26
Index
0
1
2
3

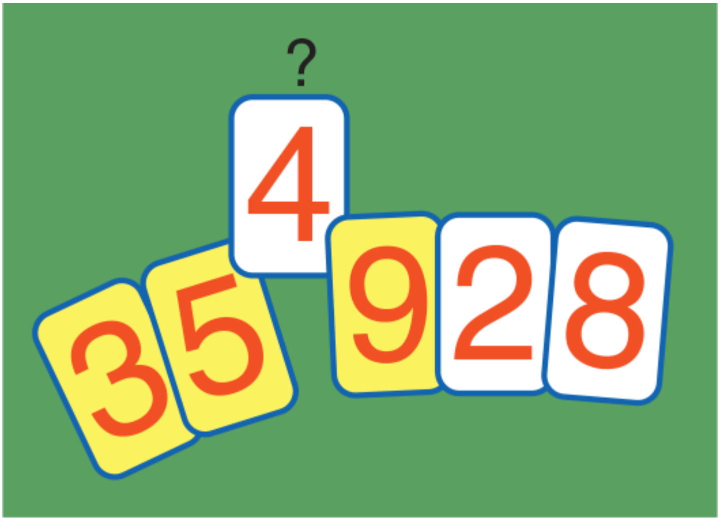
```
int temp = array[3];        // line 1
array[3] = array[6];        // line 2
array[6] = temp;            // line 3
```

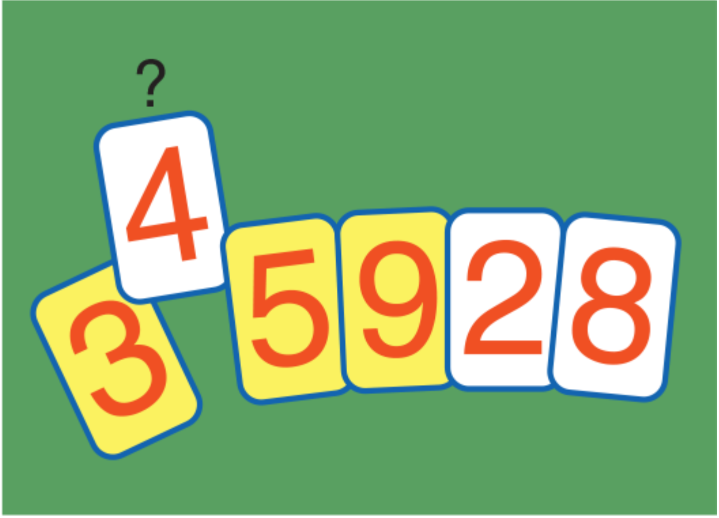
Value	23	45	7	33	78	90	82	80	90	66
Index	0	1	2	3	4	5	6	7	8	9

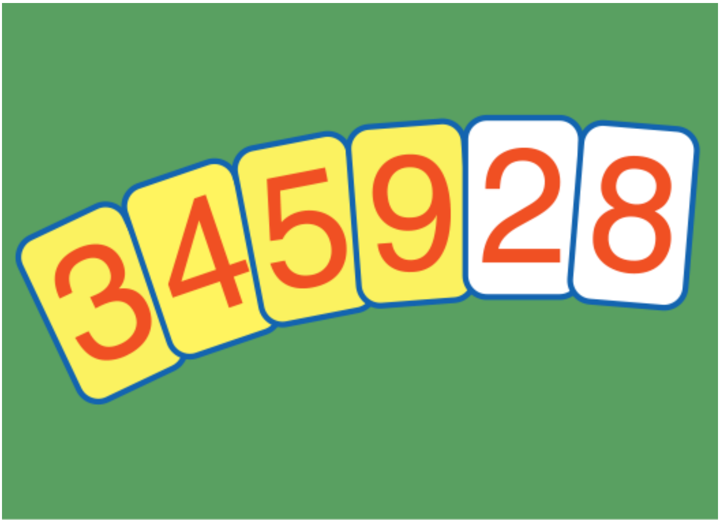
Value	23	45	7	33	78	90	82	80	90	66		33
Index	0	1	2	3	4	5	6	7	8	9		temp
Value	23	45	7	82	78	90	82	80	90	66		33
Index	0	1	2	3	4	5	6	7	8	9		temp
Value	23	45	7	82	78	90	33	80	90	66		33
Index	0	1	2	3	4	5	6	7	8	9		temp

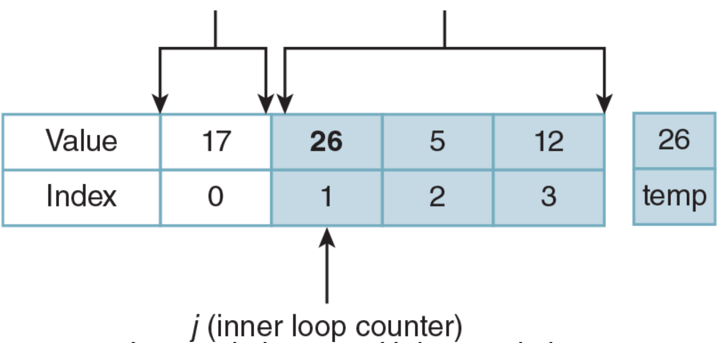
```
Before Selection Sort, the array is
3394    279    1181    2471    3660    221

After Selection Sort, the array is
221     279    1181    2471    3394    3660
```



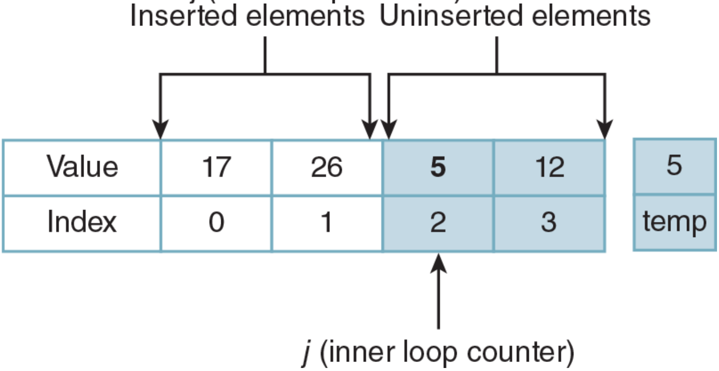

?
4
3
5
9
2
8


?
4
3
5
9
2
8

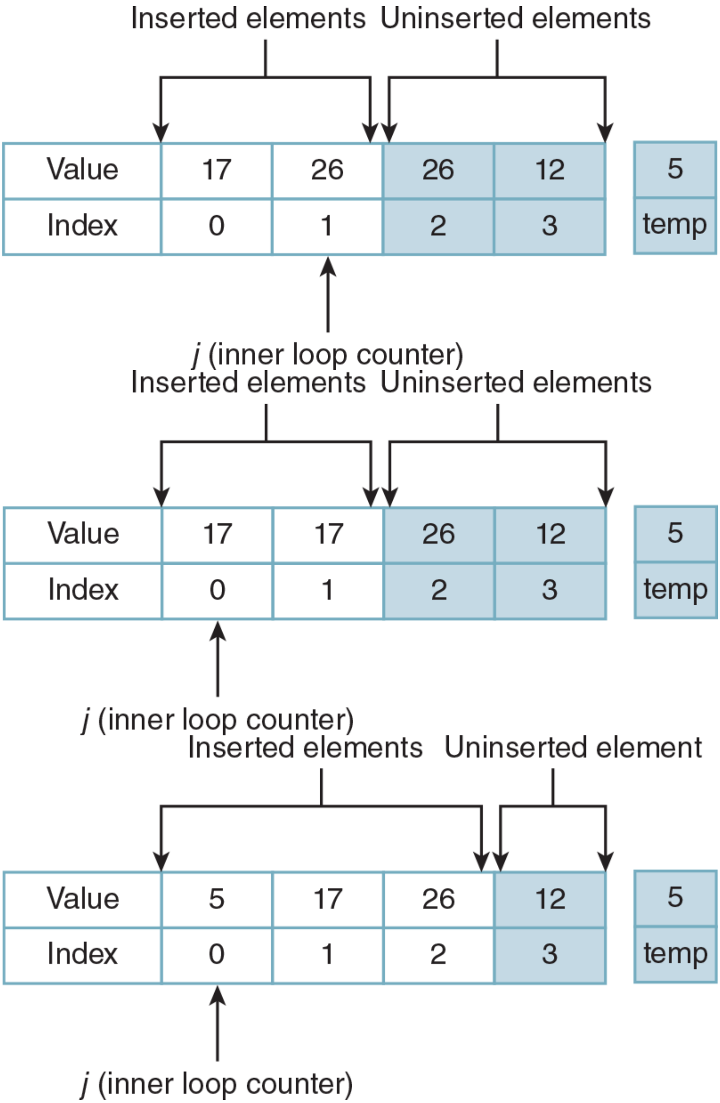


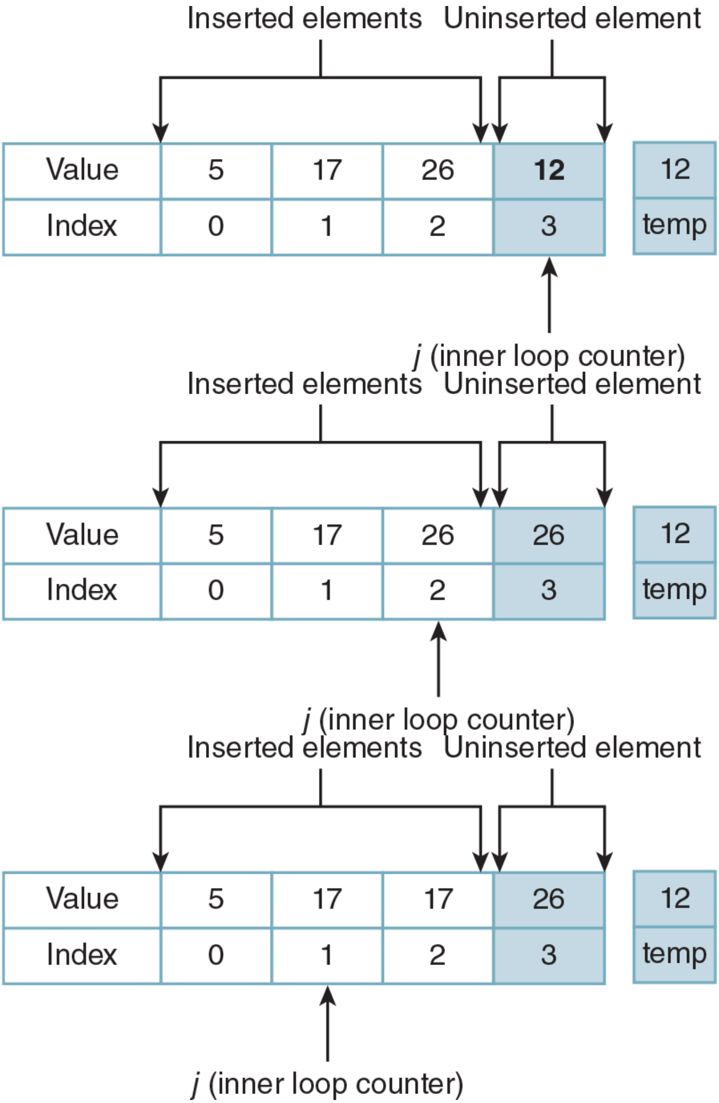

Inserted element
Uninserted elements
j (inner loop counter)
26
temp

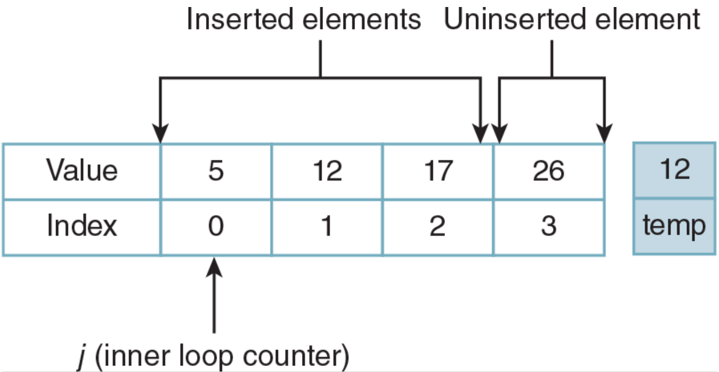
Value	17	**26**	5	12
Index	0	1	2	3

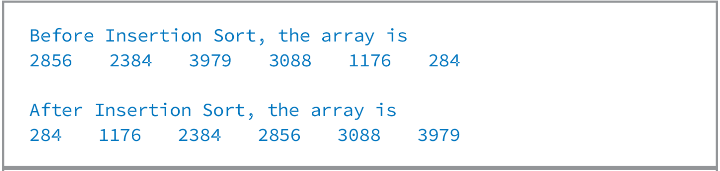

Inserted elements
Uninserted elements
j (inner loop counter)
5
temp

Value	17	26	**5**	12
Index	0	1	2	3

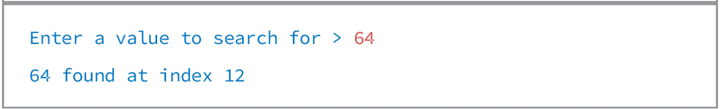

Inserted elements
Uninserted elements
Value
Index
temp
j (inner loop counter)
Inserted elements
Uninserted elements
Value
Index
temp
j (inner loop counter)
Inserted elements
Uninserted element
Value
Index
temp
j (inner loop counter)


Inserted elements
Uninserted element
Value
5
17
26
12
Index
0
1
2
3
12
temp
j (inner loop counter)
Inserted elements
Uninserted element
Value
5
17
26
26
Index
0
1
2
3
12
temp
j (inner loop counter)
Inserted elements
Uninserted element
Value
5
17
17
26
Index
0
1
2
3
12
temp
j (inner loop counter)

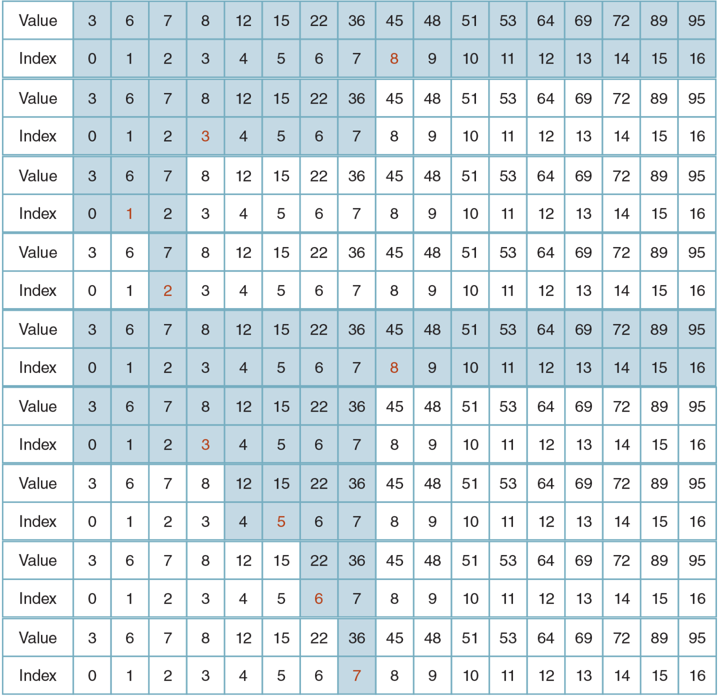




```
Before Insertion Sort, the array is
2856    2384    3979    3088    1176    284

After Insertion Sort, the array is
284    1176    2384    2856    3088    3979
```



```
Enter a value to search for > 64

64 found at index 12
```



Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

8.6 Searching and Sorting Arrays

Arrays are great instruments for storing a large number of related values. As seen earlier in this chapter, we can use arrays to store daily temperatures, CD titles, telephone bills, quiz grades, and other sets of related values. Once the data is stored in an array, we will want to manipulate that data. A very common operation is searching an array for a specific value.

8.6.1 Sequential Search of an Unsorted Array

Let's assume we manage a movie theater. We give each customer a frequent moviegoer card with a unique member ID. We have decided to pick four member IDs at random and give those members a free gift the next time they visit the theater. So we set up a *MovieWinners* class with two array instance variables:

- An array of *ints* that holds the member IDs of the winners
- An array of *Strings* that holds the corresponding prizes

Note that both arrays have four elements and that there is a one-to-one correspondence between the two arrays. Winner #1 will receive prize #1, winner #2 will receive prize #2, and so on. This programming technique is called using **parallel arrays**.

We fill the *winners* array with member IDs chosen randomly. We fill the *prizes* array with *Strings* representing prize descriptions. When a member buys a movie ticket, we look through the *winners* array for the member's ID. If the member's ID is in the *winners* array, we use its index to retrieve the corresponding prize from the *prizes* array that the member won. If the member ID is not found in the array, we know the member is not a winner.

The *MovieWinners* class is shown in **Example 8.14**.

Example 8.14 The *MovieWinners* Class

```java
/** Winners of Free Movies and Other Prizes
*   Anderson, Franceschi
*/

import java.util.Random;

public class MovieWinners
{
   public final int MEMBERS = 5000;
   // array to hold winning member numbers chosen at random
   private int [ ] winners;
   // parallel array that holds prizes
   private String [ ] prizes = { "2 free movie tickets!",
                                 "1 free movie ticket!",
                                 "free popcorn!",
                                 "free box of candy!" };
   /** Default constructor instantiates winners array
   * and randomly generates winning member IDs
   */
   public MovieWinners( )
   {
     winners = new int [prizes.length];
     fillWinners( ); // generate winner member IDs
   }

   /** Utility method generates winner member IDs
   *     and stores them in the winners array
   */
   private void fillWinners( )
   {
     Random rand = new Random( );
     for ( int i = 0; i < winners.length; i++ )
     {
       winners[i] = rand.nextInt( MEMBERS ) + 1;
     }
   }

   /** Calls indexOfWinner with the member number
   *   then translates return value into the prize won
   *   @param memberNumber value to find
   *   @return prize
   */
   public String getPrize( int memberNumber )
   {
     int prizeIndex = indexOfWinner( memberNumber );
     if ( prizeIndex == -1 )
         return "Sorry, member is not a winner.";
     else
         return "You win " + prizes[prizeIndex];
   }

   /** Performs sequential search of winners array
   *   @param key member ID to find in winners array
   *   @return index of key if found, -1 if not found
   */
   private int indexOfWinner( int key )
   {
     for ( int i = 0; i < winners.length; i++ )
     {
        if ( winners[i] == key )
           return i;
     }
     return -1;
   }

   /** Returns printable version of MovieWinners object
   *   @return winning numbers separated by a tab
   */
  @Override
  public String toString( )
  {
     String returnValue = "";
     for ( int i = 0; i < winners.length; i++ )
     {
       returnValue += winners[i] + "\t";
     }
     return returnValue;
   }
}
```

Description

The constructor randomly generates values to fill the array by calling the utility method, *fillWinners* (lines 26–36). In the interest of keeping things simple, we have coded the *fillWinners* method in such a way that it does not necessarily generate different numbers; however, the likelihood of two winning numbers being equal is very small. We declare the *fillWinners* method as *private* because it is designed to be called only by the methods of this class.

Our *indexOfWinner* method (lines 52–64) performs a **Sequential Search**, which compares the member ID to each element in the array one by one. The *indexOfWinner* method accepts a parameter, *key*, which is the member ID to search for in the array. If *key* is found, *indexOfWinner* returns the index of that array element. If *key* is not found, that is, if none of the elements in the array matches the value of *key*, *indexOfWinner* returns −1. Since −1 is not a valid array index, it’s a good value to use to indicate that the search was unsuccessful.

Notice that if the current array element matches the *key*, the *indexOfWinner* method returns immediately to the caller (line 61); that is, the method stops executing. The return value is the index of the element that matched the *key*. If, however, the method finishes executing all iterations of the *for* loop, then the method has looked at every element in the array without finding a match. In that case, the method returns −1 (line 63), indicating that the *key* was not found.

Our *getPrize* method (lines 38–50) calls *indexOfWinner* to check if its *memberNumber* parameter is a winning number; if it is, it uses the array index returned by *indexOfWinner* to return the corresponding element in the array *prizes* (line 49).

Example 8.15 shows a client application that uses our *MovieWinners* class.

Example 8.15 Client Application for the *MovieWinners* Class

```
 1 /** Client for the MovieWinners class
 2     Anderson, Franceschi
 3 */
 4 import java.util.Scanner;
 5
 6 public class MovieWinnersClient
 7 {
 8   public static void main( String [ ] args )
 9   {
10     // instantiate the winningIDs array
11     MovieWinners winningIDs = new MovieWinners( );
12
13     // prompt for the member ID
14     Scanner scan = new Scanner( System.in );
15     System.out.print( "Enter the member's ID "
16                           + "or 0 to stop > " );
17     int searchID = scan.nextInt( );
18
19     while ( searchID != 0 )
20     {
21       // determine whether member is a winner
22       System.out.println( winningIDs.getPrize( searchID ) );
23
24       System.out.print( "\nEnter the next member's ID "
25                             + "or 0 to stop > " );
26       searchID = scan.nextInt( );
27     }
28
29     System.out.println( "\nThe winners were "
30                            + winningIDs.toString( ) );
31   }
32 }
```

Description

SOFTWARE ENGINEERING TIP

When you write a class that uses corresponding lists of items with different data types, consider using parallel arrays.

We instantiate a *MovieWinners* object reference named *winningIDs* (lines 10–11). We then prompt for a member ID (lines 15–17) and call the *getPrize* method (line 22) in order to output any prize that may have been won by the current member. **Figure 8.21** shows a possible output of running the *MovieWinnersClient* application.

Figure 8.21 Output of Example 8.15

Description

8.6.2 Selection Sort

The member IDs in the preceding *winners* array were in random order, so when a member was not a winner, our *findWinners* method needed to look at every element in the array before discovering that the ID we were looking for was not in the array. This is not efficient, since most members are not winners. The larger the array, the more inefficient a sequential search becomes. We could simplify the search by arranging the elements in numeric order, which is called **sorting the array**. Once the array is sorted, we can use various algorithms to speed up a search. Later in this chapter, we discuss how to search a sorted array.

In this chapter, we present two basic sorting algorithms, **Selection Sort** and **Insertion Sort**.

Selection Sort derives its name from the algorithm used to sort the array. We select the largest element in the array and place it at the end of the array. Then we select the next-largest element and put it in the next-to-last position in the array. To do this, we consider the unsorted portion of the array as a **subarray**. We repeatedly select the largest value in the current subarray and move it to the end of the subarray, then consider a new subarray by eliminating the elements that are in their sorted locations, until the subarray has only one element. At that time, the array is sorted.

In more formal terms, we can state the Selection Sort algorithm, presented here in pseudocode, in this way:

```
To sort an array with n elements in ascending order:
1. Consider m elements as a subarray with m = n elements.
2. Find the index of the largest value in this subarray.
3. Swap the values of the element with the largest value and the
   element in the last position in the subarray.
4. Consider a new subarray of m = m - 1 elements by eliminating
   the last element in the previous subarray.
5. Repeat steps 2 through 4 until m = 1.
```

Description

For example, let’s walk through a Selection Sort on the following array. At the beginning, the entire array is the subarray (shown here with shading).

We begin by considering the entire array as an unsorted subarray. We find that the largest element is 26 at index 1.

Description

Next we move element 1 to the last element by swapping the values of the elements at indexes 1 and 3.

The value 26 is now in the right place, and we consider elements 0 through 2 as the unsorted subarray.

Description

The largest element in the new subarray is 17 at index 0. So we move element 0 to the last index of the subarray (index 2) by swapping the elements at indexes 0 and 2.

The value 17 is now in the right place, and we consider elements 0 and 1 as the new unsorted subarray.

Description

The largest element in the new subarray is 5 at index 0. We move element 0 to the last index of the subarray (index 1) by swapping the elements at indexes 0 and 1.

The value 5 is now in the right place, and we consider element 0 as the new subarray. But because there is only one element in the subarray, the subarray is sorted. Thus the whole array is sorted, and our job is done.

Description

A critical operation in a Selection Sort is swapping two array elements. Before going further, let’s examine the algorithm for swapping two array elements.

To swap two values, we need to define a temporary variable that is of the same data type as the values being swapped. This variable will temporarily hold the value of one of the elements, so that we don’t lose the value during the swap.

The algorithm, presented here in pseudocode, involves three steps:

```
To swap elements a and b:
1. Assign the value of element a to the temporary variable.
2. Assign the value of element b to element a.
3. Assign the value in the temporary variable to element b.
```

Description

For instance, if an *array* named *array* has *int* elements, and we want to swap the element at index 3 with the element at index 6, we will use the following code:

Description

The order of these operations is critical; changing the order might result in loss of data and erroneous data stored in the array.

The following illustrates line by line what happens during the swap:

Before line 1 is executed, our array looks like this:

Description

Line 1 assigns the value of element 3 to *temp*. After line 1 is executed, the value of *temp* is 33. The array is unchanged.

Description

Line 2 assigns the value of element 6 (82) to element 3. After line 2 is executed, both element 6 and element 3 have the same value. But that's OK, because we saved the value of element 3 in *temp*.

Description

COMMON ERROR TRAP

When swapping elements, be sure to save a value before replacing it with another value to avoid losing data.

Line 3 assigns the value we saved in *temp* to element 6. After line 3 is executed, the values of elements 3 and 6 have been successfully swapped.

Description

Example 8.16 shows the *Sorter* class, which provides a *static selectionSort* method for an integer array.

Example 8.16 The *Sorter* Class

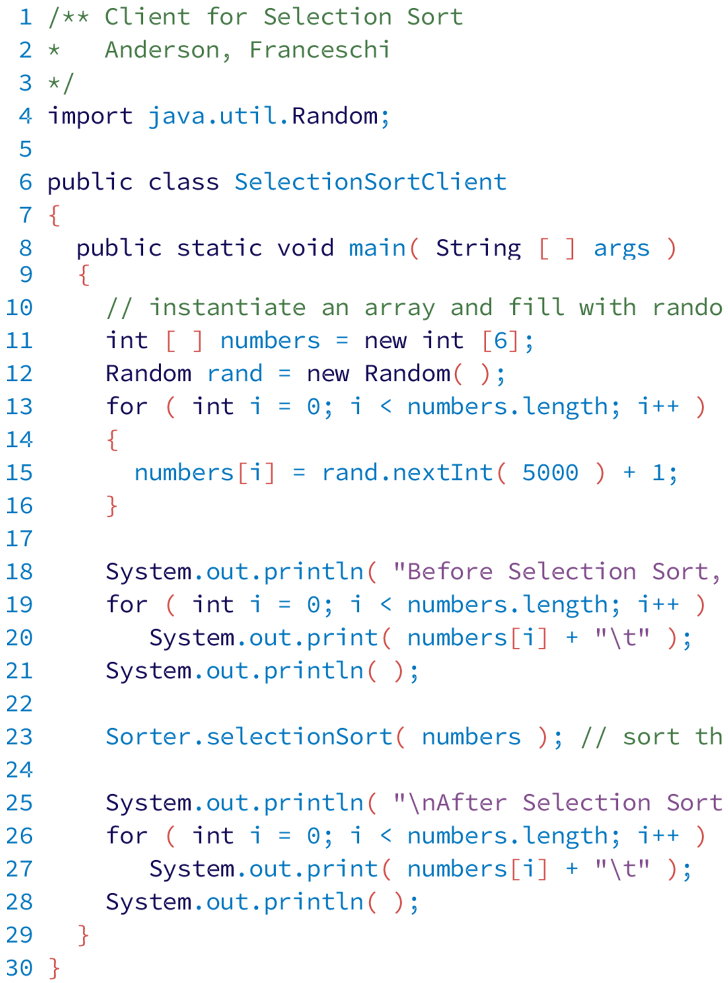
```
 1 /* Sort Utility Class
 2 *  Anderson, Franceschi
 3 */
 4
 5 public class Sorter
 6 {
 7   /**  Uses Selection Sort to sort
 8   *      an integer array in ascending order
 9   *    @param array the array to sort
10   */
11   public static void selectionSort( int [ ] array )
12   {
13     int temp; // temporary location for swap
14     int max; // index of maximum value in subarray
15
16     for ( int i = 0; i < array.length - 1; i++ )
17     {
18       // find index of largest value in subarray
19       max = indexOfLargestElement( array, array.length - i );
20
21       // swap array[max] and array[array.length - i - 1]
22       temp = array[max];
23       array[max] = array[array.length - i - 1];
24       array[array.length - i - 1] = temp;
25     }
26   }
27
28   /**  Finds index of largest element
29   *    @param    size the size of the subarray
30   *    @param    array the array to search
31   *    @return   the index of the largest element in the subarray
32   */
33   private static int indexOfLargestElement( int [ ] array, int size )
34   {
35     int index = 0;
36     for ( int i = 1; i < size; i++ )
37     {
38        if ( array[i] > array[index] )
39           index = i;
40     }
41     return index;
42   }
43 }
```

Description

Part of the Selection Sort algorithm is finding the index of the largest element in a subarray, so we implement the Selection Sort with two methods. At lines 7–26 is the *selectionSort* method, which implements the Selection Sort algorithm. To perform its work, the *selectionSort* method calls the utility method, *indexOfLargestElement* (lines 28–42), which returns the index of the largest element in a subarray. This method uses the algorithm discussed earlier in the chapter for finding a maximum value in an array. We declare this method *private* because its only function is to provide a service to the *selectionSort* method. The *indexOfLargestElement* method must also be declared as *static* because the *selectionSort* method is *static*, and thus can call only *static* methods.

In **Example 8.17**, the client code instantiates an integer array and prints the array before and after the Selection Sort is performed. Because *selectionSort* is a *static* method, we call it using the *Sorter* class name. The output of a sample run is shown in **Figure 8.22**.

Example 8.17 Using Selection Sort



```java
/** Client for Selection Sort
*   Anderson, Franceschi
*/
import java.util.Random;

public class SelectionSortClient
{
  public static void main( String [ ] args )
  {
    // instantiate an array and fill with rando
    int [ ] numbers = new int [6];
    Random rand = new Random( );
    for ( int i = 0; i < numbers.length; i++ )
    {
      numbers[i] = rand.nextInt( 5000 ) + 1;
    }

    System.out.println( "Before Selection Sort,
    for ( int i = 0; i < numbers.length; i++ )
       System.out.print( numbers[i] + "\t" );
    System.out.println( );

    Sorter.selectionSort( numbers ); // sort th

    System.out.println( "\nAfter Selection Sort
    for ( int i = 0; i < numbers.length; i++ )
       System.out.print( numbers[i] + "\t" );
    System.out.println( );
  }
}
```

Description

Figure 8.22 Using Selection Sort

Description

8.6.3 Insertion Sort

Like Selection Sort, Insertion Sort derives its name from the algorithm used to sort the array. The basic approach to an Insertion Sort is to sort elements much like a card player arranges the cards in sorted order in his or her hand. The player inserts cards one at a time in such a way that the cards on the left side of his or her hand are sorted at all times; the cards on the right side of his or her hand have not yet been inserted into the sorted part of the hand. As **Figure 8.23a** shows, the three yellow cards on the left (3, 5, and 9) are already arranged in sorted order, and the white cards on the right (4, 2, and 8) have yet to be inserted into their correct location. Note that the "sorted" yellow cards on the left side are not necessarily in their final position yet. We will now insert the 4. We first compare it to the 9; since 4 is smaller than 9, we shift the 9 to the right (Figure 8.23b). We then compare the 4 to the 5; since 4 is smaller than 5, we shift the 5 to the right (Figure 8.23c). We then compare the 4 to the 3; since 4 is larger than 3, the 3 stays in place and we insert the 4 in the empty slot (Figure 8.23d). We are now ready to insert the next card, the 2.

Figure 8.23a The next card to insert is a 4

Description

Figure 8.23b 9 is shifted to the right

Description

Figure 8.23c 5 is shifted to the right

Description

Figure 8.23d 4 is inserted

Description

To sort an array of *n* elements in ascending order, Insertion Sort implements a double loop:

- The outer loop executes $n - 1$ times and iterates through the array elements from indexes 1 through $n - 1$. If the variable *i* represents the counter of the outer loop, the array can be thought of as made of three parts:
    - a sorted subarray (although the elements may not be in their final position yet) from index 0 to $i - 1$,

  - the array element (at index *i*) that we are currently inserting, and
  - a subarray (from index *i* + 1 to *n* – 1) of elements that have not yet been inserted.

- At each iteration of the outer loop, we insert the current array element at its proper place within the sorted subarray. The inner loop compares the current array element to the elements of the sorted array from right to left and shifts these elements to the right until it finds the proper insert location.
- After all elements have been inserted, the array is sorted.

The pseudocode for the Insertion Sort is

```
for i = 1 to last array index by 1
     j = i
     temp = element at index i
     while ( j != 0 and value of element at index j - 1 > temp)
           shift element at index j - 1 to the right
           decrement j by 1
     assign value stored in temp to element at index j
```

Description

For example, let's walk through an Insertion Sort on the following array. At the beginning, the unsorted array is

Description

The first element of the array, 17, is automatically in the correct position when we consider the subarray as consisting of that element only. The value of the outer loop counter (*i*) is 1, and we will now insert the second array element, 26, into the left subarray. First, we save the value of the element to be inserted by storing it in *temp*. We need to save the value because it is possible that we will shift other values, in which case we would overwrite that element. The value of the inner loop counter (*j*) is set to the value of the outer loop counter (*i*), i.e., 1. We compare elements 26 (index *j* = 1) and 17 (index *j* – 1 = 0). Since 26 is larger than 17, we exit the inner loop (and therefore we do not shift 17 to the right). We then assign the value of the current element, 26, stored in *temp*, to the element at index *j* = 1; in this case, there is no change to the array. The value 26 has been inserted.

The outer loop counter (*i*) is incremented, and its value is 2. We will now insert the third array element, 5, into the left subarray (at this point comprised of the two inserted elements, 17 and 26).

Description

The value of the inner loop counter (*j*) is set to the value of the outer loop counter (*i*), i.e., 2. We compare the current element, 5, stored in *temp*, and 26 (index *j* – 1 = 1). Since 5 is smaller than 26, we shift 26 to the right and decrement *j* by 1; *j* now has the value 1.

Description

We then compare the current element, 5, stored in *temp*, and 17 (index *j* – 1 = 0). Since 5 is smaller than 17, we shift 17 to the right and decrement *j* by 1; *j* now has the value 0.

Description

Since *j* is 0, we exit the inner loop and assign the value of the current element, 5, stored in *temp,* to the array element at index *j* = 0. The value 5 has now been inserted.

Description

The outer loop counter (*i*) is incremented, and its value is 3. We will now insert the fourth array element, 12, into the left subarray (at this point comprising the three inserted elements, 5, 17, and 26).

Description

The value of the inner loop counter (*j*) is set to the value of the outer loop counter (i), i.e., 3. We compare the current element, 12, stored in *temp,* and 26 (index *j* – 1 = 2). Since 12 is smaller than 26, we shift 26 to the right and decrement *j* by 1; *j* now has the value 2.

Description

We then compare the current element, 12, stored in *temp,* and 17 (index *j* – 1 = 1). Since 12 is smaller than 17, we shift 17 to the right and decrement *j* by 1; *j* now has the value 1.

Description

We then compare the current element, 12, stored in *temp,* and 5 (index *j* – 1 = 0). Since 12 is not smaller than 5, we exit the inner loop; we then assign the value of the current element, stored in *temp,* to the element at index *j* = 1; the value 12 has been inserted.

Description

The outer loop counter (*i*) is incremented, and its value is 4, which causes the outer loop to terminate. All the elements have been inserted; the array is now sorted.

COMMON ERROR TRAP

When looping through an array, be careful not to access an element outside the bounds of the array. Your code will compile, but will generate an *ArrayIndexOutOfBoundsException* at run time.

Example 8.18 shows our *Sorter* class with the Insertion Sort algorithm implemented in lines 43–63.

Example 8.18 *Sorter* Class with Insertion Sort

```
/* Sort Utility Class
   Anderson, Franceschi
*/

public class Sorter
{
  /**  Performs a Selection Sort on
  *      an integer array
  *    @param the array to sort
 */
 public static void selectionSort( int [ ] array )
 {
    int temp; // temporary location for swap
    int max; // index of maximum value in subarray

    for ( int i = 0; i < array.length - 1; i++ )
    {
      // find index of largest value in subarray
      max = indexOfLargestElement( array, array.length - i );

      // swap array[max] and array[array.length - i - 1]
      temp = array[max];
      array[max] = array[array.length - i - 1];
      array[array.length - i - 1] = temp;
    }
  }

  /**  Finds index of largest element
  *    @param     size the size of the subarray
  *    @return    the index of the largest element in the subarray
  */
  private static int indexOfLargestElement( int [ ] array, int size )
  {
     int index = 0;
     for ( int i = 1; i < size; i++ )
     {
         if ( array[i] > array[index] )
            index = i;
     }
     return index;
  }

  /**  Performs an Insertion Sort on an integer array
  *    @param array array to sort
  */
  public static void insertionSort( int [ ] array )
  {
    int j, temp;

    for ( int i = 1; i < array.length; i++ )
    {
       j = i;
       temp = array[i];

       while ( j != 0 && array[j - 1] > temp )
       {
          array[j] = array[j - 1];
          j--;
       }

       array[j] = temp;
    }
  }
}
```

Description

Example 8.19 shows a client program that instantiates an integer array, fills it with random values, and then prints the array before and after performing the Insertion Sort. **Figure 8.24**. shows a sample run, using the Insertion Sort algorithm to sort an array of integers.

Example 8.19 Using Insertion Sort

```
/** Client for Insertion Sort
*   Anderson, Franceschi
*/
import java.util.Random;

public class InsertionSortClient
{
  public static void main( String [ ] args )
  {
    // instantiate an array and fill with random values
    int [ ] numbers = new int [6];
    Random rand = new Random( );
    for ( int i = 0; i < numbers.length; i++ )
    {
      numbers[i] = rand.nextInt( 5000 ) + 1;
    }

    System.out.println( "Before Insertion Sort, the array is" );
    for ( int i = 0; i < numbers.length; i++ )
       System.out.print( numbers[i] + "\t" );
    System.out.println( );

    Sorter.insertionSort( numbers ); // sort the array

    System.out.println( "\nAfter Insertion Sort, the array is" );
    for ( int i = 0; i < numbers.length; i++ )
         System.out.print( numbers[i] + "\t" );
    System.out.println( );
  }
}
```

Description

Figure 8.24 Using Insertion Sort

Description

8.6.4 Sorting Arrays of Objects

We saw earlier in the chapter that data items to be sorted can be primitive data types, such as integers or *doubles*. But they can also be objects. With an array of objects, it is important to understand that we need to sort the objects themselves, not the array elements, which are merely the object references, or memory locations of the objects.

Arrays of objects are sorted using a sort key, which is one or more of the instance variables of the objects. For instance, if we have email objects, they can be sorted by date received, by author, by subject, and so on. It is important to note that when we sort objects, the integrity of the objects must be respected; for instance, when we sort a collection of email objects by sender, we sort a collection of email objects, not a collection of senders.

Thus, to perform the Insertion Sort on the *cars* array of *Auto* objects, we need to decide which field (or fields) of the *Auto* object determines the order of the objects. If we say that the *model* is the sort field, then the comparison statement would compare the models in two objects, that is, two *Strings*. The *compareTo* method of the *String* class compares the values of two *Strings.* It returns a positive number if the *String* for which the method is invoked is greater than the *String* passed as an argument.

To sort the *cars* array using an Insertion Sort, we would need to make several revisions to the *InsertionSort* method. First, the data type of the array must be declared as an *Auto* in the parameter list. Second, *temp* needs to be defined as an *Auto* reference, and finally, we need to substitute the *compareTo* method in the condition that compares array elements.

The revised Insertion Sort code becomes:

```
   /* * Insertion sorts an array of Autos
   *      @param arr an array of Autos
   */
   public static void insertionSort( Auto [ ] arr )
   {
      Auto temp;
      int j;

   for ( int i = 1; i < arr.length; i++ )
   {
       j = i;
       temp = arr[i];

       while ( j != 0 && ( temp.getModel( ) ).compareTo(
               arr[j - 1].getModel( ) ) < 0 )
       {
            arr[j] = arr[j - 1];
            j--;
       } // end while loop

       arr[j] = temp;

   } // end for loop
}    // end InsertionSort method
```

Description

8.6.5 Sequential Search of a Sorted Array

Earlier in the chapter, the *MovieWinners* class sequentially searched an array. The algorithm assumed the elements were not in order. If we sort the array, a Sequential Search can be implemented more efficiently for the case when the search key is not present in the array. Instead of searching the entire array before discovering that the search key is not in the array, we can stop as soon as we pass the location where that element would be if it were in the array. In other words, if the array is sorted in ascending order, we can recognize an unsuccessful search when we find an element in the array that is greater than the search key. Because the array is sorted in ascending order, all the elements after that array element are larger than that element, and therefore are also larger than the search key.

To implement this algorithm, we can add another test to the *for* loop condition, so that we exit the loop as soon as we find an element that is greater than the search key. The improved algorithm shown next could be used to replace the *indexOfWinner* method shown in Example 8.14 for Sequential Search of a sorted *winners* array:

```
public int indexOfWinner( int key )
{
  for ( int i = 0; i < winners.length && winners[i] <= key; i++ )
  {
    if ( winners[i] == key )
    return i;
  }

  return -1; // end of array reached without finding key
             // or an element larger than the key was found
}
```

Description

In fact, if the array is sorted, it can be searched even more efficiently using an algorithm called Binary Search, which we explain in the next section.

8.6.6 Binary Search of a Sorted Array

If you've played the "Guess a Number" game, you probably have used the concept of a **Binary Search.** In this game, someone asks you to guess a secret number between 1 and 100. For each number you guess, they tell you whether the secret number is larger or smaller than your guess. A good strategy is to guess the number in the middle, which in this example is 50. Whether the secret number is larger or smaller than 50, you will have eliminated half of the possible values. If the secret number is greater than 50, then you know your next guess should be 75 (halfway between 50 and 100). If the secret number is less than 50, your next guess should be 25 (halfway between 1 and 50). If you continue eliminating half the possible numbers with each guess, you will quickly guess the secret number. This approach works because we are "searching" a sorted set of numbers (1 to 100).

Similarly, a Binary Search of a sorted array works by eliminating half the remaining elements with each comparison. First, we look at the middle element of the array. If the value of that element is the search key, we return its index. If, however, the value of the middle element is greater than the search key, then the search key cannot be found in elements with array indexes higher than that element. Therefore, we will search the left half of the array only. Similarly, if the value of the middle element is lower than the search key, then the search key cannot be found in elements with array indexes lower than the middle element. Therefore, we will search in the right half of the array only. As we keep searching, the subarray we search keeps shrinking in size. In fact, the size of the subarray we search is cut in half at every iteration.

If the search key is not in the array, the subarray we search will eventually become empty. At that point, we know that we will not find our search key, and we return –1.

Example 8.20 shows our Binary Search algorithm.

Example 8.20 Binary Search of a Sorted Array

```
 1 /** Binary Search
 2 *   Anderson, Franceschi
 3 */
 4
 5  import java.util.Scanner;
 6
 7  public class BinarySearcher
 8  {
 9    public static void main( String [ ] args )
10    {
11      // define an array sorted in ascending order
12      int [ ] numbers = { 3, 6, 7, 8, 12, 15, 22, 36, 45,
13                         48, 51, 53, 64, 69, 72, 89, 95 };
14
15      Scanner scan = new Scanner( System.in );
16      System.out.print( "Enter a value to search for > " );
17      int key = scan.nextInt( );
18
19      int index = binarySearch( numbers, key );
20      if ( index != -1 )
21           System.out.println( key + " found at index " + index );
22      else
23           System.out.println( key + " not found" );
24    }
25
26    public static int binarySearch( int [ ] arr, int key )
27    {
28      int start = 0;
29      int end = arr.length - 1;
30      int middle;
31
32      while ( end >= start )
33      {
34        middle = ( start + end ) / 2; // element in middle of array
35
36        if ( arr[middle] == key )
37        {
38           return middle; // key found at middle
39        }
40        else if ( arr[middle] > key )
41        {
42           end = middle - 1; // search left side of array
43        }
44        else
45        {
46           start = middle + 1; // search right side of array
47        }
48      }
49      return -1;
50    }
51 }
```

Description

We start by declaring and initializing an integer array with 17 sorted elements (lines 12–13). We then prompt the user for a search key and call the *binarySearch* method (lines 16–19).

The *binarySearch* method is coded at lines 26–50. The local variables *start* and *end* store the first and last index of the subarray to search. Because we begin by searching the entire array, we initialize these to the indexes of the first and last element of the array that was passed as a parameter. The local variable *middle,* declared at line 30, will store the index of the middle element in the subarray to search.

The search is performed in a *while* loop (lines 32–48), whose condition determines whether the subarray is empty. If the subarray is not empty, we calculate the value for *middle* by adding the indexes of the first and last elements and dividing by 2 (line 34). Next we test whether the value at the *middle* index is equal to the key. If so, we have found the key and we return its index, which is *middle* (lines 36–39). If not, we test whether the value in the middle of the subarray is greater than the key. If so, if the key is in the array, it will be found in the left half of the array. Thus, we reduce the subarray to the elements with indexes less than *middle* (lines 40–43) and greater than or equal to *start.* If the value in the middle of the subarray is less than the key, then if the key is in the array, it will be found in the right half of the array. Thus, we reduce the subarray to the elements with indexes greater than *middle* (lines 44–47) and smaller than or equal to *end.*

When the *while* loop continues, we continue making our comparisons and either returning the index of the search key or reducing the size of the subarray. If the search key is not in the array, the subarray eventually becomes empty, and we exit the *while* loop and return –1 (line 49). **Figure 8.25** shows the output when the search key is found.

Figure 8.25 Output from Example 8.20

Description

Let’s run through the Binary Search algorithm on the key 7 to illustrate how the algorithm works when the key is found in the array. Here is the array *numbers:*

Description

When the *binarySearch* method is called, it sets *start* to 0 and *end to arr.length – 1,* which is 16. Thus, the value of *middle* is 8.

The element at index 8 (45) is greater than 7, so we set *end* to 7 (*middle* – 1), and we will now search the left subarray, highlighted next. The value of middle is now 3 ((0 + 7) / 2).

Description

The element at index 3 (8) is greater than 7, so we set *end* to 2 (*middle* – 1) and keep searching in the left subarray, highlighted next. The value of *middle* is now 1 ((0 + 2) / 2).

Description

The element at index 1 (6) is smaller than 7, so we set *start* to 2 (*middle* + 1) and search in the right subarray, highlighted next. The value of *middle* is now 2 ((2 + 2) / 2).

Description

The element at index 2 (7) is equal to 7. We have found the value and return its index, 2.

Let's now run the preceding example on the key 34 to illustrate how the algorithm works when the key is not found in the array.

Here is the array *numbers* again:

Description

Again, when the *binarySearch* method is called, it sets *start* to 0 and *end* to *arr.length* – 1, which is 16. Thus, *middle* is assigned the value 8 for the first comparison.

The element at index 8 (45) is greater than 34, so we set *end* to 7 (*middle* – 1), and keep searching in the left subarray. The value of *middle* becomes 3 for the next comparison.

Description

The element at index 3 (8) is smaller than 34, so we search in the right subarray highlighted below. The value of *middle* is now 5.

Description

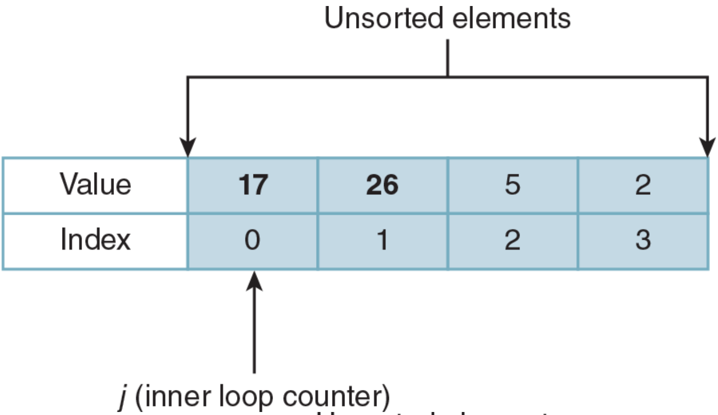
The element at index 5 (15) is smaller than 34, so we search in the right subarray. The value of *middle* is now 6.

Description

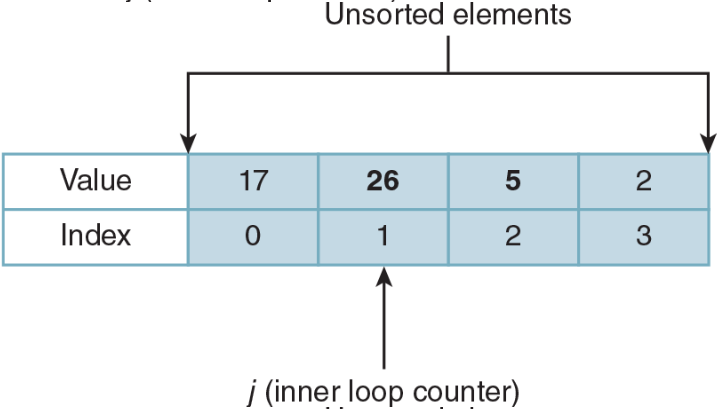
The element at index 6 (22) is smaller than 34, so we search in the right subarray. The value of *middle* is now 7.

Description

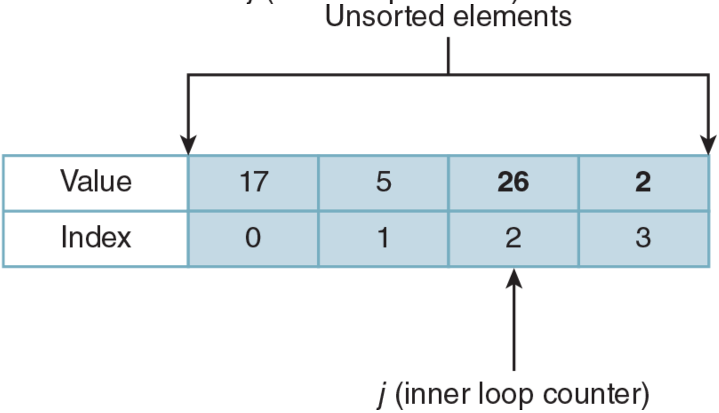
At this point, *start, end,* and *middle* all have the value 7. The element at index 7 (36) is larger than 34, so we assign *end* the value *middle* – 1, which is 6. This makes *end less* than *start* and consequently makes the while loop condition evaluate to *false*. We have not found 34, so we return –1.


Unsorted elements
j (inner loop counter)

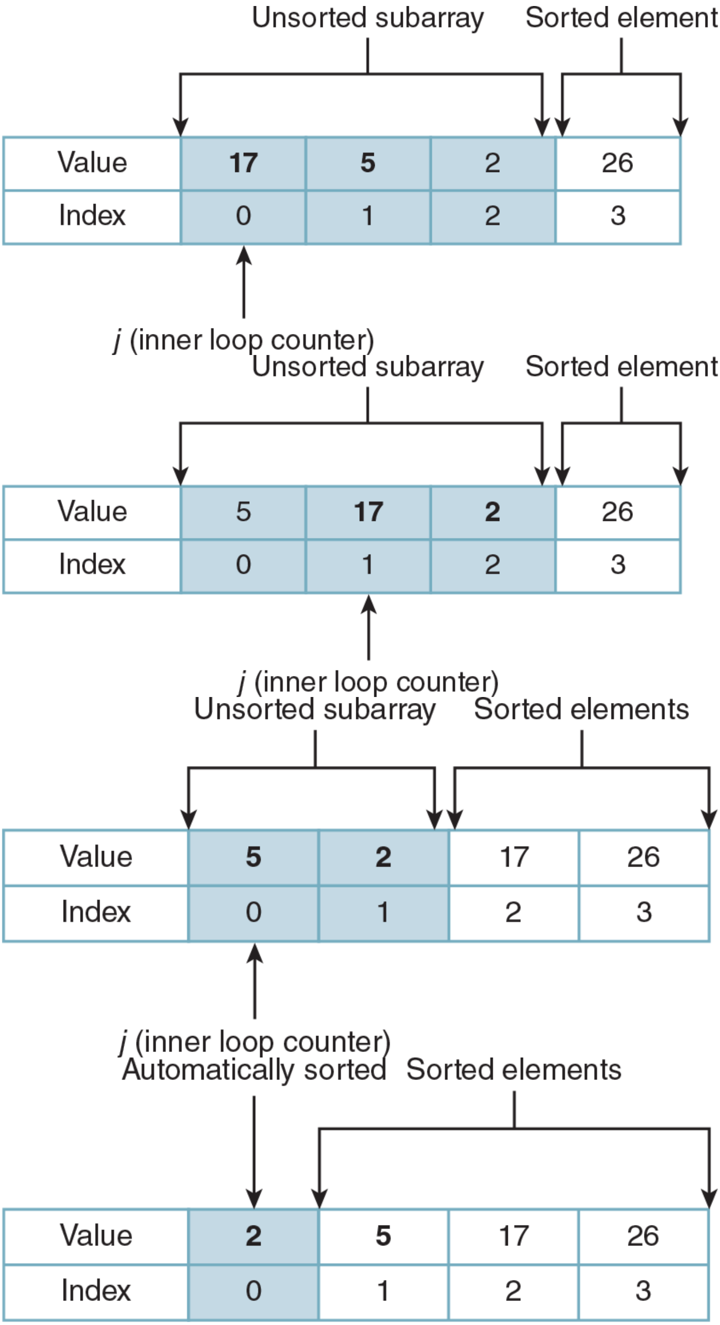
Value	**17**	**26**	5	2
Index	0	1	2	3


Unsorted elements
j (inner loop counter)

Value	17	**26**	**5**	2
Index	0	1	2	3


Unsorted elements
j (inner loop counter)

Value	17	5	**26**	**2**
Index	0	1	2	3


Unsorted subarray
Sorted element

Value	**17**	**5**	2	26
Index	0	1	2	3

j (inner loop counter)
Unsorted subarray
Sorted element

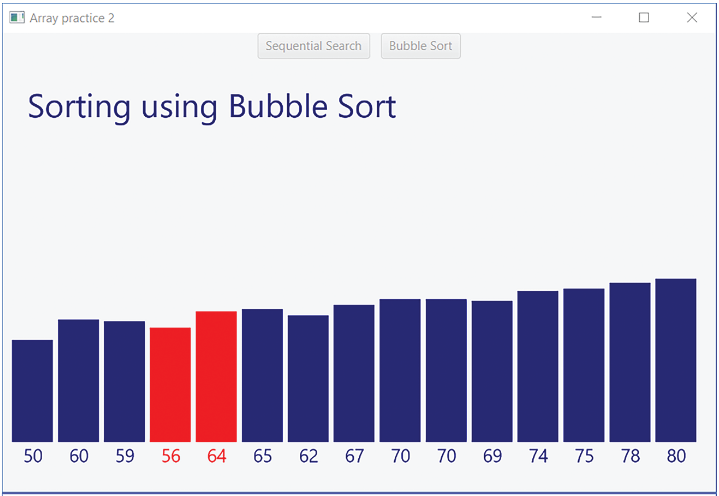
Value	5	**17**	**2**	26
Index	0	1	2	3

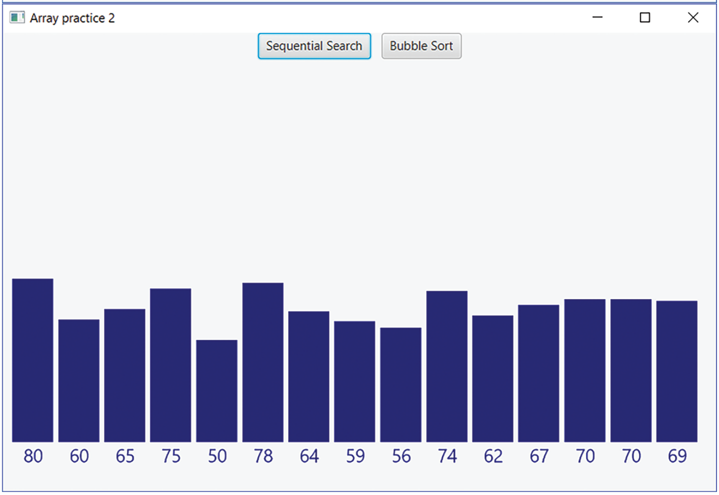
j (inner loop counter)
Unsorted subarray
Sorted elements

Value	**5**	**2**	17	26
Index	0	1	2	3

j (inner loop counter)
Automatically sorted
Sorted elements

Value	**2**	**5**	17	26
Index	0	1	2	3


Array practice 2
Sequential Search
Bubble Sort
Sorting using Bubble Sort
50
60
59
56
64
65
62
67
70
70
69
74
75
78
80


Array practice 2
Sequential Search
Bubble Sort
80
60
65
75
50
78
64
59
56
74
62
67
70
70
69

8.7 Programming Activity 2: Searching and Sorting Arrays

In this activity, you will work again with a 15-element integer array, performing these activities:

1. Write a method to perform a Sequential Search of an array.
2. Write a method to implement the Bubble Sort algorithm to sort an array.

The basic approach to a Bubble Sort is to make multiple passes through the array. In each pass, we compare adjacent elements. If any two adjacent elements are out of order, we put them in order by swapping their values.

To sort an array of n elements in ascending order, Bubble Sort implements a double loop:

- The outer loop executes $n - 1$ times.
- For each iteration of the outer loop, the inner loop steps through all the unsorted elements of the array and does the following:
  - Compares the current element with the next element in the array.
  - If the next element is smaller, it swaps the two elements.

At this point, $n - 1$ elements have been moved to their correct positions. That leaves only the element at index 0, which is therefore automatically at the correct position within the array. The array is now sorted.

As the outer loop counter goes from 0 to $n - 2$, it iterates $n - 1$ times.

Outer loop counter	**Indexes of element(s) at the sorted position**
0	$n - 1$
1	$n - 2, n - 1$
2	$n - 3, n - 2, n - 1$
…	…
$n - 3$	$2, 3, 4, \ldots, n - 3, n - 2, n - 1$
$n - 2$	$1, 2, 3, 4, \ldots, n - 3, n - 2, n - 1$

Description

The pseudocode for the Bubble Sort is

```
for i = 0 to last array index - 1 by 1
  for j = 0 to ( last array index - i -1 ) by 1
    if (2 consecutive elements are in the wrong order)
      swap them
```

Description

For example, let’s walk through a Bubble Sort on the following array. At the beginning, the unsorted array is

Description

The value of the outer loop counter (*i*) is 0, and the value of the inner loop counter (*j*) is also 0. We compare elements 17 (index *j* = 0) and 26 (index *j* + 1 = 1). Since 17 is smaller than 26, we do not swap them.

The inner loop counter (*j*) is incremented, and its value is now 1.

Description

We compare elements 26 (index *j* = 1) and 5 (index *j* + 1 = 2). Since 26 is larger than 5, we swap them. The inner loop counter (*j*) is incremented, and its value is now 2.

Description

We compare elements 26 (index *j* = 2) and 2 (index *j* + 1 = 3). Since 26 is larger than 2, we swap them.

The inner loop counter (*j*) is incremented, and its value is now 3; therefore, we exit the inner loop. (We have reached the end of the unsorted subarray, which at this point is the whole array.) At the end of one execution of the inner loop, the value 26 has “bubbled up” to its correct position within the array.

We now go back to the outer loop, and the outer loop counter (*i*) is incremented; its value is now 1. We reenter the inner loop, and the value of the inner loop counter (*j*) is reinitialized to 0.

Description

We compare elements 17 (index *j* = 0) and 5 (index *j* + 1 = 1). Since 17 is larger than 5, we swap them. The inner loop counter (*j*) is incremented, and its value is now 1.

Description

We compare elements 17 (index *j* = 1) and 2 (index *j* + 1 = 2). Since 17 is larger than 2, we swap them.

The inner loop counter (*j*) is incremented, and its value is now 2; therefore, we exit the inner loop. (We have reached the end of the unsorted subarray.) At this point, the element 17 has "bubbled up" to its correct position within the array.

We go back to the outer loop, and the outer loop counter (*i*) is incremented; its value is now 2, and this will be the last iteration of the outer loop. We reenter the inner loop, and the value of the inner loop counter (*j*) is reinitialized to 0.

Description

We compare elements 5 (index *j* = 0) and 2 (index *j* + 1 = 1). Since 2 is smaller than 5, we swap them.

The inner loop counter (*j*) is incremented, and its value is now 1; therefore, we exit the inner loop. (We have reached the end of the unsorted subarray.) At this point, the element 5 has "bubbled up" to its correct position within the array.

We go back to the outer loop, and the outer loop counter (*i*) is incremented; its value is now 3, and therefore, we exit the outer loop. For the four elements in the array, we executed the outer loop three times.

Description

The array is now sorted.

The framework for this Programming Activity will animate your algorithm so that you can watch your algorithm work and check the accuracy of your code. For example, **Figure 8.26** demonstrates the Bubble Sort at work. At this point, the program has completed four passes through the array and just finished comparing the values of elements 3 and 4.

Figure 8.26 The Bubble Sort at Work

Description

Instructions

In this chapter's Programming Activity 2 folder in the supplied code files, you will find the source files needed to complete this activity. Copy all the files to a folder on your computer. Note that all files should be in the same folder.

Open the *ArrayPractice2Controller.java* source file. Searching for five asterisks (*****) in the source code will position you at the two locations where you will add your code. Your first task is to complete the *sequentialSearch* method, which searches the *arr* array, an instance variable of the *ArrayPractice2Controller* class. The array *arr* has already been instantiated for you and filled with random values. The second task is to complete the *bubbleSort* method. **Example 8.21** shows the section of the *ArrayPractice2Controller* source code where you will add your code. Note that in each method, you are asked to call the *animate* method so that your method code can be animated as it works. Note also that for the *sequentialSearch* method, we provide a dummy *return* statement (*return 0;*). We do this so that the source code will compile. In this way, you can write and test each method separately, using stepwise refinement. When you are ready to write the *sequentialSearch* method, just replace the dummy *return* statement with the appropriate *return* statement for that method.

Example 8.21 Student Section of *ArrayPractice2Controller*

```
// 1. ***** student writes this method
/**  Searches for key in integer array named arr
//     arr is an instance variable of the class and has been
//     instantiated and filled with random values.
// @param key value to search for
// @return if key is found, the index of the first element
//     in array whose value is key; if key is not found,
//     the method returns -1
*/
public int sequentialSearch( int key )
{
 // Note:  To animate the algorithm, put this method call as the
 // first statement in your for loop
 //  animate( i );
 //         where i is the index of the current array element


   return 0; // replace this statement with your return statement

} // end of sequentialSearch
//  2. *****  student writes this method
/**  Sorts arr in ascending order using the bubble sort algorithm
*/
public void bubbleSort( )
{
 // Note: To animate the algorithm, put this method call as the
 // last statement in your innermost for loop
 //  animate( i, j );
 //           where i is the value of the outer loop counter
 //           and j is the value of the inner loop counter,
 //           or the index of the current array element

} // end of bubbleSort
```

Description

When you have finished writing your code, compile *ArrayPractice2Controller*.java and run *ArrayPractice2Application*. **Figure 8.27** shows the graphics window when the application begins. To test any method, click on the appropriate button.

Figure 8.27 Opening Window of *ArrayPractice2Application*

Description

Troubleshooting

If your method implementation does not animate, consider these tips:

- Verify that your *for* loop calls the *animate* method as instructed in the method comments.
- Verify that you did not instantiate a new array. Perform all operations on the instance variable array named *arr.*

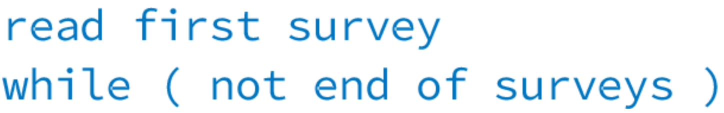
DISCUSSION QUESTIONS

1. The sequential search finds only the first occurrence of the parameter *key*. How would you modify the *sequentialSearch* method to count the occurrences of *key*?
2. It is possible that the array might be completely sorted before all the passes have been completed. How would you modify your code so that you exit the *bubbleSort* method as soon as possible?

```
Roll      Count
1         81
2         82
3         89
4         78
5         87
6         83
```

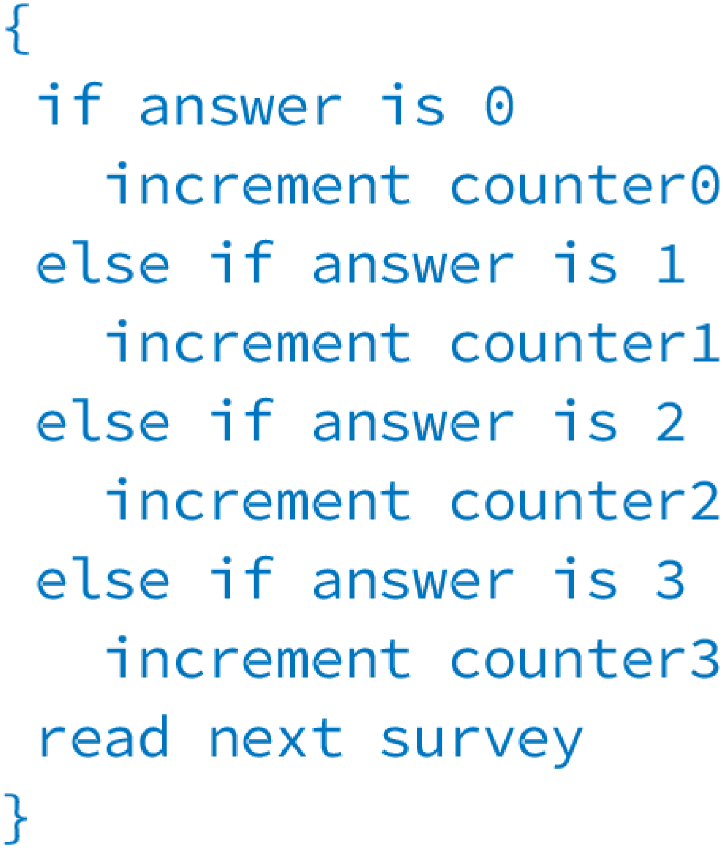
8.8 Using Arrays as Counters

In some circumstances, it is useful to use an array of integers as an ordered group of accumulators, or counters. For example, suppose we are analyzing a survey that has four possible answers, 0 through 3. We want to count how many people selected each answer. We could set up four counters and use an *if/else* statement to increment the appropriate counter. The pseudocode would be:



```
read first survey
while ( not end of surveys )
```

Description



```
{
  if answer is 0
    increment counter0
  else if answer is 1
    increment counter1
  else if answer is 2
    increment counter2
  else if answer is 3
    increment counter3
  read next survey
}
```

Description

That would work if we have only a few possible answers, but what if we had 100 or more answers? We would end up writing a very long *if/else* statement.

Instead, we could set up an array of counters and let the counter for answer 0 be *array[0]*, the counter for answer 1 be *array[1],* and so on. This approach—using an array of counters—is simpler to code and saves processing time.

As another example, suppose we want to throw a die 500 times and count the number of times each outcome occurs; that is, we want to count the number of ones, twos, threes, fours, fives, and sixes that are rolled. To do this, we set up a simple *Die* class shown in **Example 8.22**, with a method for rolling a value. Then we set up the client class, *DieCount,* shown in

Example 8.23, that has an array with six integer elements; each element will hold the number of times a particular roll occurs.

Example 8.22 The *Die* Class

```
/** Die class
*   Anderson, Franceschi
*/
import java.util.Random;

public class Die
{
    public final int SIDES = 6;
    private Random rand;

    /** default constructor
    * instantiates the Random object
    */
    public Die( )
    {
       rand = new Random( );
    }

    /** rolls the die
    * @return the value of the roll
    */
    public int roll( )
    {
       return rand.nextInt( SIDES ) + 1;
    }
}
```

Description

Example 8.23 The *DieCount* Class

```
 1 /** DieCount Class
 2 *   Anderson, Franceschi
 3 */
 4
 5 public class DieCount
 6 {
 7   public static void main( String [ ] args )
 8   {
 9      final int FACES = 6, NUMBER_OF_ROLLS = 500;
10
11      // instantiate the counter array
12      // which sets initial values to 0
13      int [ ] rollCount = new int [FACES];
14
15      // instantiate the Die
16      Die d1 = new Die( );
17
18      // roll the die 500 times
19      for ( int i = 1; i <= NUMBER_OF_ROLLS; i++ )
20      {
21         int myRoll = d1.roll( );
22         rollCount[myRoll - 1]++; // increment the counter for roll
23      }
24
25      // print count for each roll
26      System.out.println( "Roll\tCount" );
27      for ( int i = 0; i < rollCount.length; i++ )
28      {
29         System.out.println( ( i + 1 ) + "\t" + rollCount[i] );
30      }
31   }
32 }
```

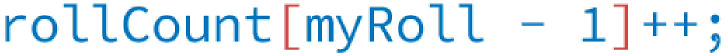
Description

In the *Die* class constructor, we instantiate the *Random* object *rand,* which will be used by the *roll* method (lines 19–25), which in turn generates a random number between 1 and 6 to simulate the roll of a die.

In the *DieCount* class, we instantiate our array of six counters, *rollCount,* on line 13, which autoinitializes each element to 0—exactly what we want for counters.

To count the number of times each roll occurs, we use a *for* loop that iterates 500 times, with each iteration calling the *roll* method of the *Die* class. We then need to count each roll. That’s where our array of counters, *rollCount,* comes in.

Since the *rollCount* array has six elements, the index of the first element is 0, and the index of the last element is 5. We will use *rollCount[0]* to hold the number of times we rolled a 1, *rollCount[1]* to hold the number of times we rolled a 2, and continue that way until we use *rollCount[5]* to hold the number of times we rolled a 6. Thus, to get the index of the appropriate counter, we need to decrement the roll by 1. So our statement to increment the count for a roll (line 22) becomes



```
rollCount[myRoll - 1]++;
```

Description

After rolling the die 500 times and counting each roll, we print the total times each roll occurred (lines 25–30). Note that we increment the loop variable to convert between our counter index and the roll number. The output from a sample run of this program is shown in **Figure 8.28**. Because the program generates the rolls randomly, your output may be slightly different.

Figure 8.28 Output from *DieCount*

Description

Our algorithm is not ideal, however. We need to subtract 1 from the index in order to increment the counter, and we need to add 1 to the index to print the outcome.

A better approach would be to create the array with seven elements. Then we can use elements 1 through 6 as the counters for the rolls 1 through 6. The index and the roll number will be the same. What happens to element 0? Nothing. We just ignore it.

The revised *DieCount2* class is shown in **Example 8.24**.

Example 8.24 The *DieCount2* Class

```
 1 /** DieCount2 Class
 2 *   Anderson, Franceschi
 3 */
 4
 5 public class DieCount2
 6 {
 7   public static void main( String [ ] args )
 8   {
 9      final int FACES = 7, NUMBER_OF_ROLLS = 500;
10
11      // instantiate the counter array
12      // which sets initial values to 0
13      int [ ] rollCount = new int [FACES];
14
15      // instantiate the Die
16      Die d1 = new Die( );
17
18      // roll the die 500 times
19      for ( int i = 1; i <= NUMBER_OF_ROLLS; i++ )
20      {
21        int myRoll = d1.roll( );
22        rollCount[myRoll]++; // increment the counter for roll
23      }
24
25      // print count for each roll
26      System.out.println( "Roll\tCount" );
27      for ( int i = 1; i < rollCount.length; i++ )
28      {
29        System.out.println( i + "\t" + rollCount[i] );
30      }
31   }
32 }
```

Description

Notice the changes to the code in this example. First, we set *FACES* to 7 (line 9), so we will instantiate an array with seven elements. Then we can use the roll of the die as the index into the counter array to increment the appropriate count (line 22). One last change is that when we loop through the *rollCount* array to print the counters, we initialize our loop counter to 1 (line 27), since we are not using element 0 as a counter and we simply use *i* as the roll number.

It’s true that we’re allocating an extra integer (four bytes of memory) that is never used, but we’re eliminating 500 subtract operations and 6 addition operations! The program is more efficient, easier to write, and easier to read.

with these end-of-chapter questions

8.11.1	Multiple Choice Exercises Question 12
8.11.4	Identifying Errors in Code Question 44
8.11.6	Write a Short Program Question 57
8.11.8	Technical Writing Question 74

```
Mary: rating 4.0
Mary: rating 4.7
Mary: rating 4.5
Mary: rating 4.5
```

8.9 Methods Accepting a Variable Number of Arguments

Sometimes, the number of arguments to be sent to a method cannot be determined until run time. For example, we may have a method that validates inputs for other methods. The number of inputs may vary from one method call to the next.

In these cases, we can use a feature in the Java language named **varargs**. We specify in the method header that the method will accept a variable number of arguments by typing three dots immediately after the data type of the last parameter.

The syntax is the following:

```
accessModifier returnType methodName( (0 to many parameters,)
                                       dataType... lastParameter )
```

Description

Note that there is no space between the data type and the three dots.

A parameter that a method specifies using the *varargs* syntax must be the last parameter listed in the method header.

When the method starts executing, the *varargs* parameters are available to the method as if the client had passed the arguments as an array. Thus, the method treats the parameter as an array.

When calling a method that accepts a variable number of arguments, we can pass a single argument, several arguments, an array of arguments, or we can pass no arguments.

Example 8.25 shows a *Seller* class with a method that accepts a variable number of arguments, and **Example 8.26** shows a client program that calls the method. In this example, we are updating the average rating a seller receives on an auction website. Each seller could have a different number of new ratings, so we specify the new ratings as a variable number of arguments.

Example 8.25 The *Seller* Class

```java
/** Seller class
*   Anderson, Franceschi
*/
import java.text.DecimalFormat;

public class Seller
{
  private String sellerName;
  private double rating;
  private int numberOfRatings;
  private static final DecimalFormat ONE_PLACE
                        = new DecimalFormat( "0.0" );

  /** constructor
    * @param sellerName seller name
  */
  public Seller( String sellerName )
  {
    this.sellerName = sellerName;
  }

  /** calcRating method
    * @param newRatings 0 to many new ratings
    * updates rating and numberOfRatings instance variables
  */
  public void calcRating( int... newRatings )
  {
    if ( newRatings != null && newRatings.length > 0 )
    {
      int totalNewRatings = 0;
      for ( int i = 0; i < newRatings.length; i++ )
        totalNewRatings += newRatings[i];

      rating = ( ( rating * numberOfRatings ) + totalNewRatings )
              / ( newRatings.length + numberOfRatings );
      numberOfRatings += newRatings.length;
    }
    // else, no new ratings, so no change to instance variables
  }

  /** toString method
   * @return the seller name
   * and rating (formatted to 1 decimal place)
  */
  @Override
  public String toString( )
  {
    return sellerName + ": rating " + ONE_PLACE.format( rating );
  }
}
```

Description

Example 8.26 The *SellerRatings* Class

```
/** SellerRatings - client for Seller
*  Anderson, Franceschi
*/

public class SellerRatings
{
  public static void main( String [ ] args )
  {
    Seller seller = new Seller( "Mary" );

    seller.calcRating( 4 ); // one value
    System.out.println( seller );

    seller.calcRating( 5, 5 ); // two values
    System.out.println( seller );

    int [ ] arrayOfRatings = { 4, 4, 5 };
    seller.calcRating( arrayOfRatings ); // array
    System.out.println( seller );

    seller.calcRating( ); // no values
    System.out.println( seller );
  }
}
```

Description

In the *Seller* class, we define three instance variables representing the seller name, the current average rating, and the number of ratings the seller has received (lines 8–10). We also define a *static* constant *ONE_PLACE* that the *toString* method (lines 41–49) uses to format the average rating.

The *calcRating* method (lines 22–39) accepts one *varargs* parameter, *newRatings*. Note that if the method accepted other parameters, the *newRatings* parameter would need to be listed last. When the method starts executing, the arguments sent to the method by the caller have been assembled into an *int* array with the parameter's name, *newRatings*. Thus, *newRatings.length* will hold the number of arguments actually sent to the method. In the *calcRating* method we first check that the array reference is not *null* and that the array is not empty (line 28). We then use a *for* loop to calculate the total of the new ratings (lines 30–32). Finally, we calculate the new average rating and update the instance variables *rating* and *numberOfRatings*. If the *varargs* parameter is *null* or the array is empty, we do nothing.

In the client program, *SellerRatings*, we instantiate a *Seller* object (line 9), then successively call the *calcRating* method with one argument, two arguments, an array, and no arguments. After each method call, we output the new rating by implicitly calling the *toString* method in the *Seller* class (lines 11–22). **Figure 8.29** shows the output from Example 8.26.

Figure 8.29 Output from the *SellerRatings* Application

Description

© Lois GoBe/Shutterstock.

CHAPTER REVIEW

8.10 Chapter Summary

- An array is a sequence of variables of the same data type. The data type can be any Java primitive data type, such as *int, float, double, byte, short, long, boolean, or char,* or it can be a class.
- Each element in the array is accessed using the array name and an index, which refers to the element's position in the array.
- Arrays are implemented as objects. Creating an array consists of declaring an object reference for the array and instantiating the array. The size of the array is given when the array is instantiated.
- In arrays of primitive types, each element of the array contains a value of that type. In arrays of objects, each element is an object reference.
- When an array is instantiated, the elements are given initial values automatically, depending on the data type. Numeric types are set to 0; *boolean* types are set to *false; char* types are set to the Unicode null character; and object references are set to *null.*
- Instantiating an array of object references involves two steps: instantiating the array and instantiating the objects.
- Arrays can be instantiated when they are declared by

assigning initial values in a comma-separated list within curly braces. The number of values in the initialization list determines the number of elements in the array.

- Array elements are accessed using the array name and an index. The first element's index is 0 and the last element's index is the size of the array –1.
- Arrays have an integer instance variable, *length,* which holds the number of elements in the array.
- Attempting to access an element of an array using an index less than 0 or greater than *arrayName.length* – 1 will generate an *ArrayIndexOutOfBoundsException* at run time.
- Aggregate array operations, such as printing and copying arrays, are not supported *for* arrays. Using a *for* loop, we can process each array element individually.
- To change the size of an array, instantiate an array of the desired size with a temporary name, copy the appropriate elements from the original array to the new array, and assign the new array reference to the original array. Assign *null* to the temporary array name.
- Arrays can be passed as arguments to methods and can also be the return type of methods.
- When an array is an instance variable of a class, the constructor should instantiate a new array and copy the elements of the parameter array into the new array.
- A Sequential Search determines whether a particular value, the search key, is in an array by comparing the search key to each element in the array.
- A Selection Sort arranges elements in the array in order by value by reducing the array into successively smaller subarrays and placing the largest element in each subarray into the last position of the subarray.
- An Insertion Sort arranges elements of an array much like a card player arranges cards in sorted order in his or her hand. The elements are inserted one at a time in ascending order into the left side of the array.
- To sort an array of objects, we can use the class method provided to compare objects' values.

- A sorted array can be searched more efficiently using a Binary Search, which successively reduces the number of elements to search by half.
- Arrays of integers can be used as an ordered group of counters.
- Methods can accept a variable number of parameters using the *varargs* … syntax.

8.11 Exercises, Problems, and Projects

8.11.1 Multiple Choice Exercises

1. What are the valid ways to declare an integer array named *a*? (Check all that apply.)

    - `int [ ] a;`
    - `int a[ ];`
    - `array int a;`
    - `int array a;`

6. What is the index of the first element of an array?

    - –1
    - 0
    - 1

10. An array *a* has 30 elements; what is the index of its last element?

    - 29
    - 30
    - 31

14. What is the default value of the elements in an array of *ints* after declaration and instantiation of the array?

    - 0
    - *null*
    - undefined

18. How do you access the element of array *a* located at index

6?

- `a{6}`
- `a( 6 )`
- `a[6]`

22. Which of the following assertions is true?

    - An array cannot be sized dynamically.
    - An array can be sized dynamically, but cannot be resized without instantiating it again.
    - An array can be sized dynamically and can also be resized without instantiating it again.

26. How do you retrieve the number of elements in an array *a*?

    - `a.length( )`
    - `a.length`
    - `a.size( )`
    - `a.size`

31. All the elements of an array must be of the same data type.

    - true
    - false

34. Array aggregate assignment is possible in Java.

    - true
    - false

37. Aggregate comparison of arrays is possible in Java.

    - true
    - false

40. An array can be returned by a method.

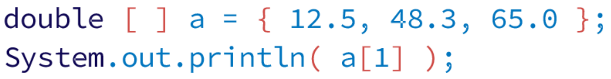
- true
- false

43. A Sequential Search on a sorted array can be written more efficiently than a Sequential Search on an unsorted array.

    - true
    - false

8.11.2 Reading and Understanding Code

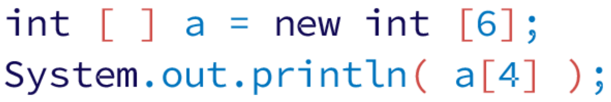
1. What is the output of this code sequence?



```
double [ ] a = { 12.5, 48.3, 65.0 };
System.out.println( a[1] );
```

Description

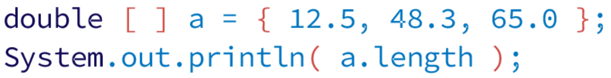
2. What is the output of this code sequence?



```
int [ ] a = new int [6];
System.out.println( a[4] );
```

Description

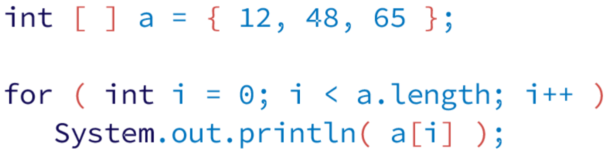
3. What is the output of this code sequence?



```
double [ ] a = { 12.5, 48.3, 65.0 };
System.out.println( a.length );
```

Description

4. What is the output of this code sequence?



```
int [ ] a = { 12, 48, 65 };

for ( int i = 0; i < a.length; i++ )
   System.out.println( a[i] );
```

Description

5. What is the output of this code sequence?

```
int [ ] a = { 12, 48, 65 };

for ( int i = 0; i < a.length; i++ )
   System.out.println( "a[" + i + "] = " + a[i] );
```

Description

6. What is the output of this code sequence?

```
int s = 0;
int [ ] a = { 12, 48, 65 };

for ( int i = 0; i < a.length; i++ )
   s += a[i];
System.out.println( "s = " + s );
```

Description

7. What is the output of this code sequence?

```
int [ ] a = new int[10];

for ( int i = 0; i < a.length; i++ )
a[i] = i + 10;

System.out.println( a[4] );
```

Description

8. What is the output of this code sequence?

```
double [ ] a = { 12.3, 99.6, 48.2, 65.8 };
double temp = a[0];

for ( int i = 1; i < a.length; i++ )
{
   if ( a[i] > temp )
      temp = a[i];
}

System.out.println( temp );
```

Description

9. What is the output of this code sequence?

```
int [ ] a = { 12, 48, 65, 23 };
int temp = a[1];
a[1] = a[3];
a[3] = temp;

for ( int i = 0; i < a.length; i++ )
  System.out.print( a[i] + " " );
```

Description

10. What does this method do?

```
public int foo( int [ ] a )
{
   int temp = 0;
   for ( int i = 0; i < a.length; i++ )
   {
         if ( a[i] == 5 )
               temp++;
   }
   return temp;
}
```

Description

11. What does this method do?

```
public int foo( int [ ] a )
{
     for ( int i = 0; i < a.length; i++ )
     {
           if ( a[i] == 10 )
                return i;
     }
     return -1;
}
```

Description

12. What does this method do?

```
public boolean foo( int [ ] a )
{
     for ( int i = 0; i < a.length; i++ )
     {
          if ( a[i] < 0 )
               return false;
     }
     return true;
}
```

Description

13. What does this method do?

```
public String [ ] foo( String [ ] a )
{
     String [ ] temp = new String[a.length];
     for ( int i = 0; i < a.length; i++ )
     {
          temp[i] = a[i].toLowerCase( );
     }
     return temp;
}
```

Description

14. What does this method do?

```
public boolean [ ] foo( String [ ] a )
{
     boolean [ ] temp = new boolean[a.length];
     for ( int i = 0; i < a.length; i++ )
     {
          if (a[i].indexOf( "@" ) != -1 )
               temp[i] = true;
          else
               temp[i] = false;
     }
     return temp;
}
```

Description

8.11.3 Fill In the Code

1. This code assigns the value 10 to all the elements of an array *a*.

   ```
   int [ ] a = new int[25];
   for ( int i = 0; i < a.length; i++ )
   {
           // your code goes here
   }
   ```

   Description

2. This code prints all the elements of array *a* that have a value greater than 20.

   ```
   double [ ] a = { 45.2, 13.1, 12.8, 87.4, 99.0, 100.1, 43.8, 2.4 };
   for ( int i = 0; i < a.length; i++ )
   {
         // your code goes here
   }
   ```

   Description

3. This code prints the average of the elements of array *a*.

   ```
   int [ ] a = { 45, 13, 12, 87, 99, 100, 43, 2 };

   double average = 0.0;
   for ( int i = 0; i < a.length; i++ )
   {
           // your code goes here
   }
   // ... and your code continues here
   ```

   Description

4. This code calculates and prints the dot product of two arrays (∑ *a*[*i*] * *b*[*i*]).

```
int [ ] a = { 3, 7, 9 };
int [ ] b = { 2, 9, 4 };
int dotProduct = 0;

for ( int i = 0; i < a.length; i++ )
{
      // your code goes here
}
```

Description

5. This code prints the following three lines:

```
a[0] = 3
a[1] = 6
a[2] = 10
```

```
int [ ] a = { 3, 6, 10 };
for ( int i = 0; i < a.length; i++ )
{
        // your code goes here
}
```

Description

6. This method returns *true* if an element in an array of *Strings* passed as a parameter contains the substring *IBM*; otherwise, it returns *false*.

```
public boolean foo( String [ ] a )
{
      // your code goes here
}
```

Description

7. This method returns the number of elements in an array passed as a parameter that are multiples of 7.

```
public int foo( int [ ] a )
{
        // your code goes here
}
```

Description

8. This method returns true if the first two elements of the array passed as a parameter have the same value; otherwise, it returns *false*.

```
public boolean foo( String [ ] a )

{
        // your code goes here
}
```

Description

9. This method takes an array of *ints* as a parameter and returns an array of *booleans*. For each element in the parameter array whose value is 0, the corresponding element of the array returned will be assigned *false*; otherwise, the element will be assigned *true*.

```
public boolean [ ] foo( int [ ] a )
{
        // your code goes here
}
```

Description

8.11.4 Identifying Errors in Code

1. Where is the error in this code sequence?

```
double [ ] a = { 3.3, 26.0, 48.4 };
a[4] = 2.5;
```

Description

2. Where is the error in this code sequence?

```
double [ ] a = { 3.3, 26.0, 48.4 };
System.out.println( a[-1] );
```

Description

3. Where is the error in this code sequence?

```
double [ ] a = { 3.3, 26.0, 48.4 };
System.out.println( a{1} );
```

Description

4. Where is the error in this code sequence?

```
double [ ] a = { 3.3, 26.0, 48.4 };
for ( int i = 0; i <= a.length; i++ )
 System.out.println( a[i] );
```

Description

5. Where is the error in this code sequence?

```
double a[3] = { 3.3, 26.0, 48.4 };
```

Description

6. Where is the error (although this code will compile and run) in this code sequence?

```
int a[ ] = { 3, 26, 48, 5 };
int b[ ] = { 3, 26, 48, 5 };

if ( a != b )
      System.out.println( "Array elements are NOT identical" );
```

Description

7. Where is the error in this code sequence?

```
int [ ] a = { 3, 26, 48, 5 };
a.length = 10;
```

Description

8. Where is the logic error in this code sequence?

```
int [ ] a = { 3, 26, 48, 5 };
System.out.println( "The array elements are " + a );
```

Description

9. Where is the error in this code sequence?

```
Integer i1 = 10;
Integer i2 = 15;
Double d1 = 3.4;
String s = new String( "Hello" );
Integer [ ] a = { i1, i2, d1, s };
```

Description

8.11.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. You coded the following on line 26 of the class *Test.java*:

```
int a[6] = { 2, 7, 8, 9, 11, 16 }; // line 26
```

Description

When you compile, you get the following messages:

```
Test.java:26: error: ']' expected
       int a[6] = { 2, 7, 8, 9, 11, 16}; // line 26
             ^
Test.java:26: error: not a statement
       int a[6] = { 2, 7, 8, 9, 11, 16}; // line 26
                    ^
Test.java:26: error: ';' expected
     int a[6] = { 2, 7, 8, 9, 11, 16}; // line 26
                   ^
3 errors
```

Description

Explain what the problem is and how to fix it.

2. You coded the following on lines 26, 27, and 28 of the class

Test.java:

```
int [ ] a = { 2, 7, 8, 9, 11, 16 }; // line 26 of class Test.java
for ( int i = 0; i <= a.length; i++ ) // line 27 of class Test.java
      System.out.println( a[i] ); // line 28 of class Test.java
```

Description

The code compiles properly, but when you run, you get the following output:

```
2
7
8
9
11
16
Exception in thread "main"
   java.lang.ArrayIndexOutOfBoundsException: Index 6 out of
   bounds for length 6 at Test.main(Test46.java:28)
```

Description

Explain what the problem is and how to fix it.

3. You coded the following in the class *Test.java*:

```
int [ ] a = { 1, 2, 3 };
int [ ] b = { 1, 2, 3 };
if ( a == b )
      System.out.println( "Arrays are equal" );
else
      System.out.println( "Arrays are NOT equal" );
```

Description

The code compiles properly and runs, but the result is not what you expected; the output is

```
Arrays are NOT equal
```

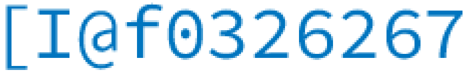
Explain what the problem is and how to fix it.

4. You coded the following in the class *Test.java*:

```
int [ ] a = { 1, 2, 3 };
System.out.println( a );
```

Description

The code compiles properly and runs, but the result is not what you expected; instead of 1 2 3, the output is similar to the following:



```
[I@f0326267
```

Description
Explain what the problem is and how to fix it.

8.11.6 Write a Short Program

1. Write a value-returning method that returns the number of elements in an integer array.
2. Write a value-returning method that returns the product of all the elements in an integer array.
3. Write a *void* method that sets to 0 all the elements of an integer array.
4. Write a *void* method that multiplies by 2 all the elements of an array of *floats*.
5. Write a method that returns the percentage of elements greater than or equal to 90 in an array of *ints*.
6. Write a method that returns the difference between the largest and smallest elements in an array of *doubles*.
7. Write a method that returns the sum of all the elements of an array of *ints* that have an odd index.
8. Write a method that returns the percentage of the number of elements that have the value *true* in an array of *booleans*.
9. Write a method that returns *true* if an array of *Strings* contains the *String “Hello”; false* otherwise.
10. Write a method that prints all the elements of an array of *chars* in reverse order.

11. Write a method that returns an array composed of all the elements in an array of *chars* in reverse order.
12. Write an array-returning method that takes a *String* as a parameter and returns the corresponding array of *chars*.
13. Code an array-returning method that takes an array of *ints* as a parameter and returns an array of *booleans*, assigning *true* for any element of the parameter array greater than or equal to 100; and *false* otherwise.

8.11.7 Programming Projects

1. Write a class encapsulating the concept of statistics for a baseball team, which has the following attributes: a number of players, a list of number of hits for each player, a list of number of at-bats for each player.
   Write the following methods:

   - A constuctor with two equal-length arrays as parameters, the number of hits per player, and the number of at-bats per player.
   - Accessors, mutators, *toString*, and *equals* methods.
   - Generate and return an array of batting averages based on the attributes given.
   - Calculate and return the total number of hits for the team.
   - Calculate and return the number of players with a batting average greater than .300.
   - A method returning an array holding the number of hits, sorted in ascending order.

   Write a client class to test all the methods in your class.
8. Write a class encapsulating the concept of student grades on a test, assuming student grades are composed of a list of integers between 0 and 100.
   Write the following methods:

- A constructor with just one parameter, the number of students; all grades can be randomly generated
- Accessor, mutator, *toString*, and *equals* methods
- A method returning an array of the grades sorted in ascending order
- A method returning the highest grade
- A method returning the average grade
- A method returning the median grade (*Hint:* The median grade will be located in the middle of the sorted array of grades.)
- A method returning the mode (the grade that occurs most often) (*Hint:* Create an array of counters; count how many times each grade occurs; then pick the maximum in the array of counters; the array index is the mode.)

Write a client class to test all the methods in your class.

16. Write a class encapsulating the concept of daily temperatures for a week.
Write the following methods:

    - A constructor accepting an array of seven temperatures as a parameter
    - Accessor, mutator, *toString*, and *equals* methods
    - A method returning how many temperatures were below freezing
    - A method returning an array of temperatures above 100 degrees
    - A method returning the largest change in temperature between any two consecutive days
    - A method returning an array of daily temperatures, sorted in descending order

    Write a client class to test all the methods in your class.

23. Write a class encapsulating the concept of a tic-tac-toe game as follows:
Two players will be playing, player 1 and player 2.

The board is represented by an array of 9 integer elements: elements at indexes 0, 1, and 2 represent the first row; elements at indexes 3, 4, and 5 represent the second row; elements at indexes 6, 7, and 8 represent the third row.

The value 0 in the array indicates that this space is available; the value 1 indicates the space is occupied by player 1; and the value 2 indicates that this space is occupied by player 2.

In the *main* method of your client class, your program will simulate a tic-tac-toe game from the command line, doing the following:

- Create a *TicTacToe* object and instantiate it.
- In a loop, prompt for plays, as *ints*, from the user. At each iteration of the loop, you will need to call methods of the *TicTacToe* class to update the *TicTacToe* object. You need to keep track of who is playing (player 1 or 2), enforce the rules, check if either player has won the game. It is clear that if anyone has won the game, it is the last player who played.
- If a player wins, you will need to exit the loop and present the result of the game. If the game ends in a tie, you should output that result.

In your *TicTacToe* class, you will need to code the following methods:

- A default constructor instantiating the array representing the board.
- A method that allows a player to make a move; it takes two arguments: the player number and the position played on the board.
- A method checking if a play is legal.
- A method checking if a player has won the game; you can break up that method into several methods if you like (for instance, check if a player has won the game by claiming an entire horizontal row).
- A method that checks whether the game is a tie (if no player has won and all squares have been played, the

game is tied).

- A method that displays the results of the game (“Player 1 won,” “Player 2 won,” or “Tie game”).

Write a client class, where the *main* method is located, to test all the methods in your class and enable the user to play.

33. When a new user logs in for the first time on a website, the user has to submit personal information, such as user_id, password, name, email address, telephone number, and so forth. Typically, there are two fields for passwords, requiring the user to enter the password twice, to ensure that the user did not make a typo in the first password field.

Write a class encapsulating the concept of processing a form with the following elements:

User_id
Password
Reenter password
Email address
Name
Street address
City
State
Zip
Telephone

In your class, write the following methods:

- A constructor with one parameter, a sequence of 10 words in an array of *Strings*, your only instance variable.
- Accessor, mutator, *toString*, and *equals* methods.
- A method checking that no *Strings* in the array are empty. (All fields are mandatory.) If at least one is empty, it returns *false*; otherwise, it returns *true*.
- A method returning the number of characters in the

user_id.

- A method checking if the two *Strings* representing the passwords (representing the password typed in twice) are identical. If they are, it returns *true*; if not, it returns *false*.
- A method checking if the *String* representing the email address actually “looks like” an email address; to simplify, we can assume that an email address contains one and only one @ character and contains one or more periods after the @ character. If it does “look like” an email address, then the method returns *true*; otherwise, it returns *false*.
- A method checking if the *String* representing the state has exactly two characters. If it does, it returns *true*; otherwise, it returns *false*.

Write a client class to test all the methods in your class.

52. We want to write a program that performs some syntax checking on HTML code; for simplicity reasons, we will assume that the HTML code is syntactically correct if the number of < characters in any word is the same as the number of > characters in that word. We will also assume that the syntax is correct if the first word is *<html>* and the last word is *</html>*.

    Write a class encapsulating that concept, including the following methods:

    - A constructor with one parameter, an array of the words in the HTML sentence, your only instance variable. Your constructor should then get user input from the console for that same number of words and store them in an array of *Strings*, your only data member.
    - Accessor, mutator, *toString*, and *equals* methods.
    - A method returning how many words are in the array.
    - A method returning *true* if the first word is *<html>* and the last word is *</html>*; false otherwise.
    - A method checking if each array element contains the same number of < characters as > characters. If that is

the case, the method returns *true*; otherwise, it returns *false*. For this, we suggest the following method to help you:

    - Write an *int*-returning method that takes a *String* and a *char* as parameters and returns how many times that *char* appears in the *String*; you can convert the *String* to an array of *chars* and loop through it, or use another strategy of your choice.

  - A method counting and returning the number of *img* tags overall.

  Write a client class to test all the methods in your class.

60. Write a class encapsulating the concept of converting integer grades to letter grades (A, B, C, D, or F), assuming grades are composed of a list of integers between 0 and 100.
    Write the following methods:

    - A constructor with just one parameter, the number of students; all grades can be randomly generated.
    - Accessor, mutator, *toString*, and *equals* methods.
    - A method returning an array of *chars* corresponding to the integer grades (90 or above should be converted to A, 80 or above to B, 70 or above to C, 60 or above to D, and 59 or less to F).
    - A method returning the number of A's.
    - A method returning an array of *ints* counting how many A's, B's, C's, D's, and F's were received.

    Write a client class to test all the methods in your class.

66. Write a class that includes a method that converts two parallel arrays of *Strings* to a *Hashtable* (look up the *Hashtable* class in the Java Class Library) such that the keys of the *Hashtable* are the elements of the first array and its corresponding values are the elements of the second array. Test your method with a client program.

67. Write a class encapsulating the concept of a team of baseball players, assuming a baseball player has the following attributes: a name, a position, and a batting percentage. In addition to that class, you will need to design and code a *Player* class to encapsulate the concept of a baseball player.

    In your class encapsulating the team, you should write the following methods:

    - A constructor taking an array of *Player* objects as its only parameter and assigning that array to the array data member of the class, its only instance variable. In your client class, when you test all your methods, you can hard-code nine baseball *Player* objects.
    - Accessor, mutator, *toString*, and *equals* methods.
    - A method checking that all positions are different, returning *true* if they are, *false* if they are not.
    - A method returning the batting percentage of the team.
    - A method checking that we have a pitcher (that is, the name of the position) on the team. If we do not have

any, it returns *false*; otherwise, it returns *true*.

- A method returning the array of *Player* objects sorted in ascending order using the batting percentage as the sorting key.
- A method checking if a certain person (a parameter of the method) is on the team, based on the name of that person. If the person is on the team, the method returns *true*; otherwise, it returns *false*.
- A method returning an array of *Player* objects, sorted in ascending order based on batting percentages.

Write a client class to test all the methods in your class.

76. Write a class encapsulating a similar concept to the one used in the die counting problem of Section 8.8. Here, we want to roll two dice; the total of the numbers rolled will be between 2 and 12. We want to keep track of how many times each possible total was rolled.
Write the following methods:

    - A constructor with no parameter; it randomly generates two numbers between 1 and 6, representing the dice.
    - Accessor, mutator, *toString*, and *equals* methods.
    - A method returning the total of the two dice.
    - A method checking if the two dice have identical values. If they do, it returns *true*; otherwise, it returns *false*.

    The number of times we roll the dice should be an input from the user at the command line (not inside the program). Your program should output the total for each possible roll (from 2 to 12), as well as the number of times the two dice had identical values.
    Write a client class to test all the methods in your class.

81. Write a graphical application that creates two *Die* objects and rolls the two dice 5,000 times. Display the results showing the frequency of each possible total in a bar chart. Pick a scale that is appropriate for the maximum height of your bar chart.

8.11.8 Technical Writing

1. What do you think are advantages and disadvantages of arrays?
2. Write the pseudocode to perform a Selection Sort on an array of *Auto* objects based on the instance variable *model*.
3. When you try to use an array index that is out of bounds, your code will compile, but you will generate a run-time exception. Discuss whether this is an advantage or a disadvantage, and why.
4. When instantiating an array, you can assign the number of elements in the array dynamically, using a variable (as opposed to using a constant). Discuss a situation where that would be useful.

8.11.9 Group Project (for a group of 1, 2, or 3 students)

1. Security is an important feature of information systems. Often, text is encrypted before being sent, and then decrypted upon receipt. We want to build a class (or several classes) encapsulating the concept of encryption. You will need to test that class with a client program where the *main* method is located.

   For this project, encrypting consists of translating each character into another character. For instance, if we consider the English alphabet, including characters *a* through *z*, each character is randomly encrypted into another, which could be the same character. (If you like, you can design your program so that no character is encrypted into itself.) To represent this concept, we can have an array of characters for the original alphabet, and another array of characters for the encrypted alphabet. For example, we could have

Original alphabet	**Encrypted alphabet**
a	u
b	p
c	h
d	a
e	s
f	x
g	z
h	b
i	j
. . .	. . .

To encrypt a word, each letter in the word is replaced by the corresponding letter in the encrypted alphabet. For example, the word *caged* would be encrypted into *huzsa*. To decrypt a word, the letters in the encrypted word are replaced by the corresponding letter in the original alphabet. For example, the encrypted word *xssa* would be decrypted as *feed*.

If we have 26 different characters in the original alphabet, then we will have 26 different characters in the encrypted alphabet. Furthermore, the encrypted alphabet should be randomly generated.

In your *main* method, you should prompt the user for a sentence. Your program should encrypt the sentence, output the encrypted sentence, then decrypt it, and output the decrypted sentence, which should be identical to the original sentence that was input by the user.

For extra credit, use an array to keep track of the number of occurrences of each character. Convert these occurrences to percentages, and then use these percentages to attempt to decrypt a large, encrypted message.

© Lois GoBe/Shutterstock.

CHAPTER 9

Multidimensional Arrays and the *ArrayList* and *HashMap* Classes

CHAPTER CONTENTS

Introduction
9.1 Declaring and Instantiating Multidimensional Arrays
  9.1.1 Declaring Multidimensional Arrays
  9.1.2 Instantiating Multidimensional Arrays
  9.1.3 Combining the Declaration and Instantiation of Multidimensional Arrays
  9.1.4 Assigning Initial Values to Multidimensional Arrays
9.2 Accessing Multidimensional Array Elements
9.3 Aggregate Two-Dimensional Array Operations
  9.3.1 Processing All the Elements of a Two-Dimensional Array
  9.3.2 Processing a Given Row of a Two-Dimensional

Array
9.3.3 Processing a Given Column of a Two-Dimensional Array
9.3.4 Processing a Two-Dimensional Array One Row at a Time
9.3.5 Processing a Two-Dimensional Array One Column at a Time
9.3.6 Displaying Two-Dimensional Array Data as a Bar Chart

9.4 Two-Dimensional Arrays Passed to and Returned from Methods
9.5 **Programming Activity 1: Working with Two-Dimensional Arrays**
9.6 Other Multidimensional Arrays
9.7 The *ArrayList* Class

9.7.1 Declaring and Instantiating *ArrayList* Objects
9.7.2 Methods of the *ArrayList* Class
9.7.3 Looping Through an *ArrayList* Using an Enhanced *for* Loop
9.7.4 Using the *ArrayList* Class in a Program

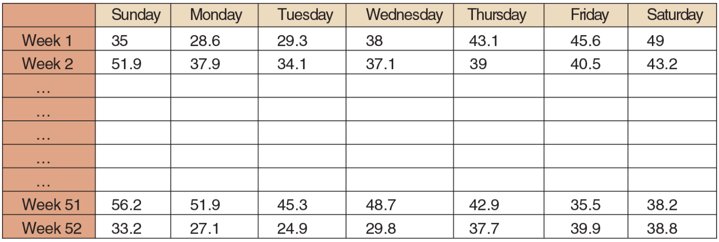
9.8 **Programming Activity 2: Working with the *ArrayList* Class**
9.9 The *HashMap* Class

9.9.1 Declaring and Instantiating *HashMap* objects
9.9.2 Methods of the *HashMap* Class
9.9.3 Using the *HashMap* in a Program
9.9.3 Using the *HashMap* in a Program

9.10 Chapter Summary
9.11 Exercises, Problems, and Projects

9.11.1 Multiple Choice Exercises
9.11.2 Reading and Understanding Code
9.11.3 Fill In the Code

9.11.4 Identifying Errors in Code
9.11.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
9.11.6 Write a Short Program
9.11.7 Programming Projects
9.11.8 Technical Writing
9.11.9 Group Project



	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Week 1	35	28.6	29.3	38	43.1	45.6	49
Week 2	51.9	37.9	34.1	37.1	39	40.5	43.2
…							
…							
…							
…							
…							
Week 51	56.2	51.9	45.3	48.7	42.9	35.5	38.2
Week 52	33.2	27.1	24.9	29.8	37.7	39.9	38.8

Introduction

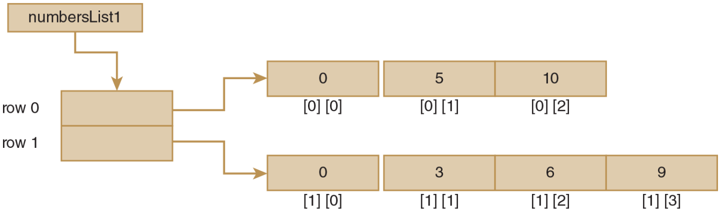
Arrays can be useful when we have a lot of data to store in memory. If we write a program to perform statistics on last year's temperatures, it is convenient to set up an array of *doubles* of size 365 to store the daily temperature data.

But what if in addition to analyzing daily temperatures, we want to analyze temperatures by the week, or by a particular day of the week? For instance, if we sail on weekends, we could want to know how many times the temperature was above 65 degrees on Saturdays and Sundays. If we are considering investing in air conditioning at home, we might be interested in knowing how many weeks had temperatures above 90 degrees. If we are avid skiers, we could be interested in the number of weeks with temperatures lower than 32 degrees.

In this situation, we would want to organize our data along two dimensions: weeks and days of the week. If we were to visualize the data as a table, we could imagine a table made up of 52 rows, each row representing a week. Each row would have seven columns, representing the days of the week. This table is shown in **Figure 9.1**. Or we could imagine a table of seven rows, each row representing a day of the week, and 52 columns, each column representing a week of the year. In either case, we can represent the rows and columns of our temperature table using a two-dimensional array. More generally, **multidimensional** arrays allow us to represent data organized along *n* dimensions with a single array.

Figure 9.1 Temperature Data for the Previous 52 Weeks

Description



9.1 Declaring and Instantiating Multidimensional Arrays

Just like single-dimensional arrays, multidimensional arrays are implemented as objects, so creating a multidimensional array takes the same two steps as creating a single-dimensional array:

1. declaring the object reference for the array
2. instantiating the array

In arrays with elements of primitive types, each element of the array contains a value of that type. For example, in an array of *doubles*, each element contains a *double* value. In arrays with a class data type, each element is an object reference, which points to the location of an object of that class.

9.1.1 Declaring Multidimensional Arrays

To declare a multidimensional array, we use the same syntax as for a single-dimensional array, except that we include an empty set of brackets for each dimension.

Here is the syntax for declaring a two-dimensional array:

```
datatype [ ][ ] arrayName;
```

Description

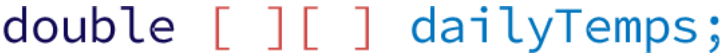
Here is the syntax for declaring a three-dimensional array:

```
datatype [ ][ ][ ] arrayName;
```

Description

In order to keep things simple, we will concentrate on two-dimensional arrays at this point. We will discuss three- and four-dimensional arrays later in the chapter.

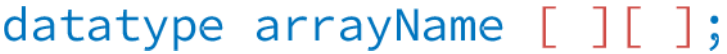
The following statement declares an array that we can use to hold the daily high temperatures for the last 52 weeks:



```
double [ ][ ] dailyTemps;
```

Description

The brackets can be placed before or after the array name. So the following syntax for declaring a two-dimensional array is also valid:

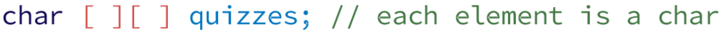


```
datatype arrayName [ ][ ];
```

Description

We prefer to put the brackets right after the data type, because it’s easier to read.

To store quiz grades for students, we could declare a two-dimensional array, where each row will store the quiz grades for a particular student and each column will store the grades for a particular quiz:

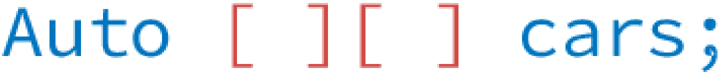


```
char [ ][ ] quizzes; // each element is a char
```

Description

The syntax is the same whether we declare arrays with basic data types or class types.

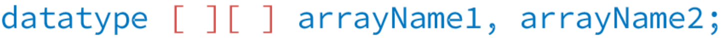
Imagine that we are interested in keeping track of a fleet of cars within a multinational corporation. The corporation operates in various countries, and in each of these countries, some employees have a company car. For this situation, we can declare a two-dimensional array where the first dimension will represent the country and the second dimension will represent the employee. Assuming we have an *Auto* class, the following statement declares this two-dimensional array to hold *Auto* objects:



```
Auto [ ][ ] cars;
```

Description

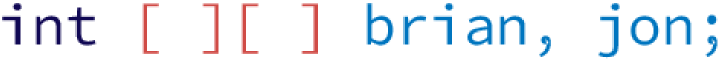
We can also declare multiple multidimensional arrays of the same data type in one statement by inserting a comma after each array name, using this syntax:



```
datatype [ ][ ] arrayName1, arrayName2;
```

Description

For example, the following statement will declare two integer arrays to hold the number of stolen bases for two baseball players for each game in their career:



```
int [ ][ ] brian, jon;
```

Description

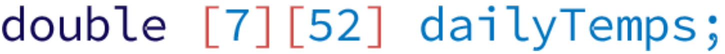
The first dimension represents the games (per season), and the second dimension represents the season.

COMMON ERROR TRAP

Specifying the size of any of the dimensions of a multidimensional array in the declaration will generate a compiler error.

Notice that when we declare a multidimensional array, we do not specify how many elements the array will have. Declaring a multidimensional array does not allocate memory for the array; this is done in step 2, when we instantiate the array.

For example, this code:



```
double [7][52] dailyTemps;
```

Description

will generate compiler errors.

9.1.2 Instantiating Multidimensional Arrays

Just like instantiating single-dimensional arrays, we instantiate a multidimensional array using the *new* keyword. Here is the syntax for instantiating a two-dimensional array:

```
arrayName = new datatype [exp1][exp2];

  where exp1 and exp2 are expressions that evaluate to integers and specify,
  respectively, the number of rows and the number of columns in the array.
```

Description

This statement allocates memory for the array. The number of elements in a two-dimensional array is equal to the sum of the number of elements in each row. When all the rows have the same number of columns, the number of elements in the array is equal to the number of rows multiplied by the number of columns.

For example, if we instantiate the following *dailyTemps* array with 52 rows and 7 columns, the array will have 52 * 7, or 364, elements:

```
dailyTemps = new double [52][7]; // dailyTemps has 52 rows
                                 // and 7 columns,
                                 // for a total of 364 elements
```

Description

These statements will instantiate the other arrays declared above:

```
int numberOfStudents = 25;
int numberOfQuizzes = 10;
quizzes = new char [numberOfStudents][numberOfQuizzes];
// quizzes has 25 rows and 10 columns
// for a total of 250 elements

cars = new Auto [5][50];
// cars has 5 rows and 50 columns
// cars will store 250 Auto objects

brian = new int [80][20];
// brian has 80 rows and 20 columns
// there are 80 games per season
// brian played baseball for 20 seasons

jon = new int [80][10];
// jon has 80 rows and 10 columns
// jon played baseball for 10 seasons
```

Description

When a multidimensional array is instantiated, the elements are given initial values automatically. Elements of arrays with numeric types are initialized to 0, elements of *char* type are initialized to the Unicode null character, elements of *boolean* type are initialized to *false*, and elements of class types are initialized to *null*.

9.1.3 Combining the Declaration and Instantiation of Multidimensional Arrays

Multidimensional arrays, like single-dimensional arrays, can also be instantiated when they are declared. To combine the declaration and instantiation of a two-dimensional array, use this syntax:

```
datatype [ ][ ] arrayName = new datatype [exp1][exp2];

  where exp1 and exp2 are expressions that evaluate to integers and specify,
  respectively, the number of rows and columns in the array.
```

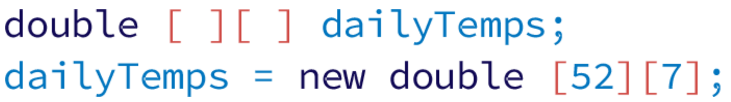
Description

Thus, this statement:

```
double [ ][ ] dailyTemps = new double [52][7];
```

Description

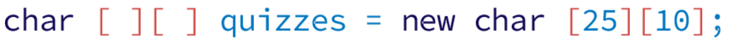
is equivalent to:



```
double [ ][ ] dailyTemps;
dailyTemps = new double [52][7];
```

Description

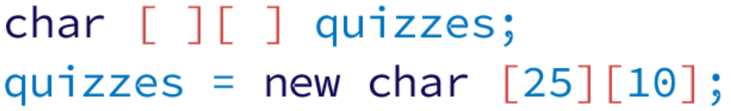
Similarly, this statement:



```
char [ ][ ] quizzes = new char [25][10];
```

Description

is equivalent to:

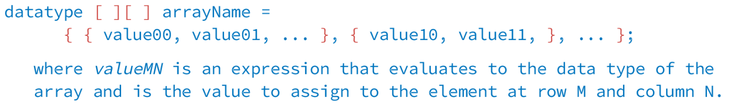


```
char [ ][ ] quizzes;
quizzes = new char [25][10];
```

Description

9.1.4 Assigning Initial Values to Multidimensional Arrays

We can instantiate a two-dimensional array by assigning initial values when the array is declared. To do this, we specify the initial values using comma-separated lists, enclosed in an outer set of curly braces:



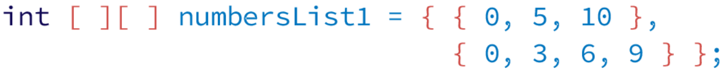
```
datatype [ ][ ] arrayName =
      { { value00, value01, ... }, { value10, value11, }, ... };

   where valueMN is an expression that evaluates to the data type of the
   array and is the value to assign to the element at row M and column N.
```

Description

The list contains a number of sublists, separated by commas. The number of these sublists determines the number of rows in the array. For each row, the number of values in the corresponding sublist determines the number of columns in the row. Thus, Java allows a two-dimensional array to have a different number of columns in each row. For example, in our *Auto* array, each country (row) could have a different number of employees (columns) with company cars.

Indeed, a two-dimensional array is an array of arrays. The first dimension of a two-dimensional array consists of an array of array references, with each reference pointing to a single-dimensional array. Thus, a two-dimensional array is composed of an array of rows, where each row is a single-dimensional array.

For example, this statement declares and instantiates a two-dimensional array of integers:



```
int [ ][ ] numbersList1 = { { 0, 5, 10 },
                            { 0, 3, 6, 9 } };
```

Description

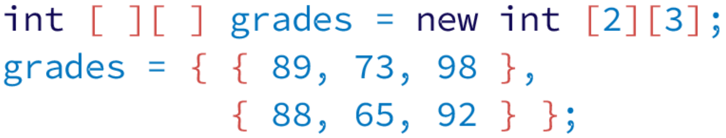
Because two sublists are given, this two-dimensional array has two rows. The first sublist specifies three values, and therefore, the first row will have three columns; the second sublist specifies four values, and therefore, the second row will have four columns.

Figure 9.2 shows the *numbersList1* array after the preceding statement is executed.

Figure 9.2 The *numbersList1* Array After Instantiation

Description

An initialization list can be given only when the array is declared. If a two-dimensional array has already been instantiated, attempting to assign values to an array using an initialization list will generate a compiler error. For example, this code:



```
int [ ][ ] grades = new int [2][3];
grades = { { 89, 73, 98 },
           { 88, 65, 92 } };
```

Description

will generate compiler errors.

We can declare and instantiate an array of objects by providing object references in the list:

```
Auto sportsCar = new Auto( "Ferrari", 0, 0.0 );
Auto sedan1 = new Auto( "BMW", 0, 0.0 );
Auto sedan2 = new Auto( "BMW", 100, 15.0 );
Auto sedan3 = new Auto( "Toyota", 0, 0.0 );
Auto rv1 = new Auto( "Jeep", 0, 0.0 );

Auto [ ][ ] cars = { { sportsCar, sedan1 },
                     { rv1, new Auto( ) },
                     { sedan2, sedan3 } };
```

Description

COMMON ERROR TRAP

An initialization list can be given only when the two-dimensional array is declared. Attempting to assign values to an array using an initialization list after the array is instantiated will generate a compiler error.

This array of *Auto* objects has three rows with two columns in each row. The elements of the array *cars* are object references to *Auto* objects.

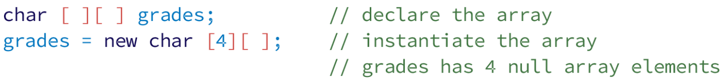
In most situations, the number of columns will be the same for each row. However, there are situations where it is useful to have a different number of columns for each row. For instance, Dr. Smith, a college professor, keeps track of grades using a two-dimensional array. The rows represent the courses she teaches and the columns represent the grades for the students in those sections. Grades are A, B, C, D, or F, so she declares the array with *char* elements. Dr. Smith teaches four courses: CS1, CS2, Database Management, and Operating Systems. Thus, she has four rows in the array. But in each course, Dr. Smith has a different number of students: There are 23 students in CS1, 16 in CS2, 12 in Database Management, and 28 in Operating Systems. So the first row will have 23 columns, the second row 16 columns, the third row 12 columns, and the fourth and last row will have 28 columns.

Using an initialization list, it is easy to instantiate a two-dimensional array with a different number of columns for every row. But sometimes the data are retrieved dynamically—read from a file, for example—and it is not possible to use an initialization list.

To instantiate a two-dimensional array with a different number of columns for each row, we can do the following:

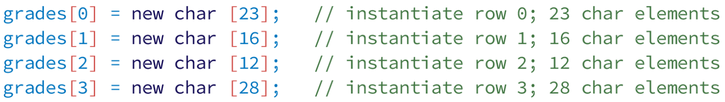
- First, instantiate the two-dimensional array.
- Second, instantiate each row as a single-dimensional array.

For the preceding example, we can use the following code:



```
char [ ][ ] grades;          // declare the array
grades = new char [4][ ];    // instantiate the array
                             // grades has 4 null array elements
```

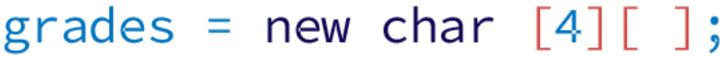
Description



```
grades[0] = new char [23];   // instantiate row 0; 23 char elements
grades[1] = new char [16];   // instantiate row 1; 16 char elements
grades[2] = new char [12];   // instantiate row 2; 12 char elements
grades[3] = new char [28];   // instantiate row 3; 28 char elements
```

Description

The second statement:



```
grades = new char [4][ ];
```

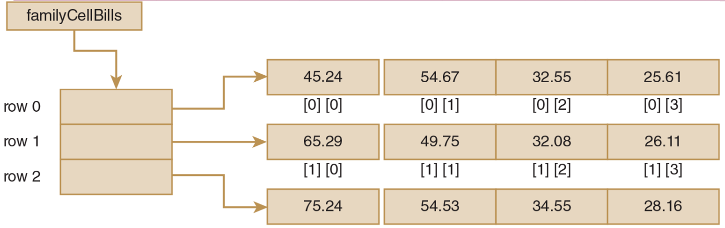
Description

instantiates the two-dimensional array *grades* as an array having four rows, none of which has been instantiated yet. Because a two-dimensional array is an array of arrays, each element of the first dimension of the *grades* array is an array reference. Thus, before being instantiated, each element of the first dimension of the *grades* array has the value *null*.

As explained earlier, in a two-dimensional array, each row is a single-dimensional array. The last four statements instantiate each row, *grades[0]*, *grades[1]*, *grades[2]*, and *grades[3]*, each row having a different number of elements, or columns. The elements in these arrays are *chars*, initialized to the Unicode null character.

Later in this chapter, we will define a general pattern for processing two-dimensional array elements so that it applies to all situations: an identical number of columns for each row, or a different number of columns for each row.

	Joe	Jane	Mike	Sarah
July	45.24	54.67	32.55	25.61
August	65.29	49.75	32.08	26.11
September	75.24	54.53	34.55	28.16


familyCellBills
row 0
row 1
row 2
45.24
[0] [0]
54.67
[0] [1]
32.55
[0] [2]
25.61
[0] [3]
65.29
[1] [0]
49.75
[1] [1]
32.08
[1] [2]
26.11
[1] [3]
75.24
[2] [0]
54.53
[2] [1]
34.55
[2] [2]
28.16
[2] [3]

```
The first monthly cell bill for the first family member is 45.24
The last monthly cell bill for the last family member is 28.16

The number of rows is 3
The number of columns in row 0 is 4
The number of columns in row 1 is 4
The number of columns in row 2 is 4
```

```
cars[1][0]'s description is:

Model: Ferrari; miles driven; 0; gallons of gas: 0.0
```

9.2 Accessing Multidimensional Array Elements

Elements of a two-dimensional array are accessed using this syntax:

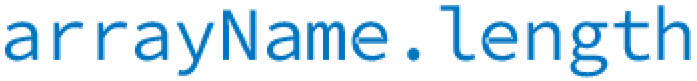
```
arrayName[exp1][exp2]
```

where *exp1* and *exp2* are expressions that evaluate to integers.

Description

Exp1 is the element's row position, or **row index**, within the two-dimensional array. *Exp2* is the element's column position, or **column index**, within the two-dimensional array. The row index of the first row is always 0; the row index of the last row is always 1 less than the number of rows. The column index of the first column is always 0. The column index of the last column is always 1 fewer than the number of columns in that row.

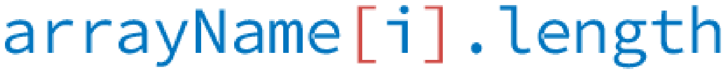
Because a two-dimensional array is an array of arrays, the length of a two-dimensional array is its number of arrays, or rows. We access the number of rows in a two-dimensional array using the following syntax:



```
arrayName.length
```

Description

Similarly, the length of each row is the number of columns (or elements) in that row's array. To access the number of columns in row *i* of a two-dimensional array named *arrayName*, we use this syntax:



```
arrayName[i].length
```

Description

Table 9.1 summarizes the syntax for accessing elements of a two-dimensional array.

Accessing Two-Dimensional Array Elements

Array Element	**Syntax**
Row 0, column *j*	`arrayName[0][j]`
Row *i*, column *j*	`arrayName[i][j]`
Last row, column *j*	`arrayName[arrayName.length - 1][j]`
Last row, last column	`arrayName[arrayName.length - 1][arrayName[arrayName.length - 1].length - 1]`
Number of rows in the array	`arrayName.length`
Number of columns in row *i*	`arrayName[i].length`

Description

Suppose we want to analyze the monthly cell phone bills for the past three months for a family of four persons. The parents, Joe and Jane, each have a cell phone, and so do the children, Mike and Sarah. We want to calculate the average monthly bill for each person, the total payments for the three months, and determine which family member had the lowest and highest bills. We could use a two-dimensional array of *doubles* with three rows and four columns. The rows will represent the months and the columns will represent the family members. For example, we could have the following mapping for the row and column indexes:

row 0 : July
row 1 : August
row 2 : September

column 0 : Joe
column 1 : Jane
column 2 : Mike
column 3 : Sarah

We could visualize our two-dimensional array as the table shown in **Table 9.2**.

Visualizing a Two-Dimensional Array

Description

We'll name the array *familyCellBills*. Each element in the array will be referenced as *familyCellBills[i][j]*, where *i* is the index of the row (the month), and *j* is the index of the column (the person). Remember that the first element in a row or column is at index 0, so the first element in the first row is at index [0][0].

In lines 13–15 of **Example 9.1**, we declare and instantiate the *familyCellBills* array. Because the elements of *familyCellBills* are *doubles*, instantiating the array also initializes each element to 0.0. Lines 18–31 store values into each element of the array. **Figure 9.3** shows how the *familyCellBills* array looks after lines 18–31 are executed.

Example 9.1 The *familyCellBills* Array

```
 1 /* Two-Dimensional Array of Cell Phone Bills
 2    Anderson, Franceschi
 3 */
 4
 5 public class FamilyCellBills
 6 {
 7  public static void main( String [ ] args )
 8  {
 9   // declare constants for the number of rows and columns
10   final int NUMBER_OF_MONTHS = 3;
11   final int NUMBER_OF_PERSONS = 4;
12
13   // declare and instantiate the array
14   double [ ][ ] familyCellBills =
15      new double [NUMBER_OF_MONTHS][NUMBER_OF_PERSONS];
16
17   // assign values to array elements
18   familyCellBills[0][0] = 45.24;  // row 0
19   familyCellBills[0][1] = 54.67;
20   familyCellBills[0][2] = 32.55;
21   familyCellBills[0][3] = 25.61;
22
23   familyCellBills[1][0] = 65.29;  // row 1
24   familyCellBills[1][1] = 49.75;
25   familyCellBills[1][2] = 32.08;
26   familyCellBills[1][3] = 26.11;
27
28   familyCellBills[2][0] = 75.24;  // row 2
29   familyCellBills[2][1] = 54.53;
30   familyCellBills[2][2] = 34.55;
31   familyCellBills[2][3] = 28.16;
32
33   System.out.println( "The first monthly cell bill for the first "
34      + "family member is\n"
35      + familyCellBills[0][0] );
36   System.out.println( "The last monthly cell bill for the last "
37      + "family member is\n"
38      + familyCellBills[NUMBER_OF_MONTHS - 1][NUMBER_OF_PERSONS - 1] );
39
40   int numRows = familyCellBills.length;
41   System.out.println( "\nThe number of rows is " + numRows );
42
43   for ( int i = 0; i < numRows; i++ )
44   {
45    System.out.print( "The number of columns in row " + i + " is " );
46    System.out.println( familyCellBills[i].length );
47   }
48  }
49 }
```

Description

Figure 9.3 The *familyCellBills* Array After Assigning Values

Description

COMMON ERROR TRAP

Attempting to access an element of a two-dimensional array using a row index less than 0 or greater than *arrayName.length – 1* will generate an *ArrayIndex-OutOfBoundsException* at run time. Similarly, attempting to access an element of row *i* of a two-dimensional array using a column index less than 0 or greater than *arrayName[i].length – 1* also will generate an *ArrayIndexOutOfBounds-Exception*.

Note that the last element is *familyCellBills[2][3]*, with a row index that is 1 less than *familyCellBills.length*, and a column index that is 1 less than *familyCellBills[2].length*, which is the number of columns in the last row. More generally, for a two-dimensional array named *arr*, the last element is:

```
arr[arr.length - 1][arr[arr.length - 1].length - 1]
```

Description

Lines 33–38 output the first and last element of the array *familyCellBills*.

Line 40 assigns the number of rows in the *familyCellBills* array to the *int* variable *numRows*. The variable *numRows* now has the value 3 and is output at line 41.

At lines 43–47, a *for* loop outputs the number of columns in each row of *familyCellBills*. **Figure 9.4** shows the output of this example.

Figure 9.4 Output of Example 9.1

Description

Row indexes of a two-dimensional array *must* be between 0 and *arrayName.length – 1*. Attempting to access an element of an array using a row index less than 0 or greater than *arrayName.length – 1* will compile without errors, but will generate an *ArrayIndexOutOfBoundsException* at run time. By default, this exception halts execution of the program.

For example, all the following expressions are invalid:

```
// invalid row indexes for the familycellBills array!!

familyCellBills[-1][2]
// the lowest valid row index is 0

familyCellBills[cellBills.length][2]
// the highest valid row index is familyCellBills.length - 1
```

Description

Similarly, column indexes of a two-dimensional array *must* be between 0 and *arrayName[i].length – 1*, where *i* is the row index. Attempting to access an element of row *i* in a two-dimensional array using a column index less than 0 or greater than *arrayName[i].length – 1* will compile without errors, but will generate an *ArrayIndexOutOfBoundsException* at run time.

For example, all the following expressions are invalid:

```
// invalid column indexes for the familyCellBills array!

familyCellBills[1][-1]
// the lowest valid column index is 0

familyCellBills[1][familyCellBills[1].length]
// the highest valid column index of row i is
// familyCellBills[i].length - 1
```

Description

Example 9.2 illustrates how to work with an array of objects. In this example, we use an *Auto* class that has three instance variables; *model*, a *String*; *milesDriven,* an *int*; and *gallonsOfGas*, a *double*. At lines 17–20, we declare and initialize *cars*, a two-dimensional array of *Auto* objects. Before using an element of *cars*, that *Auto* element has to be instantiated; failure to do so could generate a *NullPointerException* at run time.

There are three rows in *cars*: the first row has three columns, and the second and third rows have two columns each. Line 22 retrieves the array element at row 1 and column 0—here *sportsCar*—and assigns it to the *Auto* object reference *retrievedCar*, which is then printed at lines 25–26, where *toString* is called implicitly. **Figure 9.5** shows the output of this example.

Figure 9.5 Output of Example 9.2

Description

Example 9.2 Two-Dimensional Array of *Auto* Objects

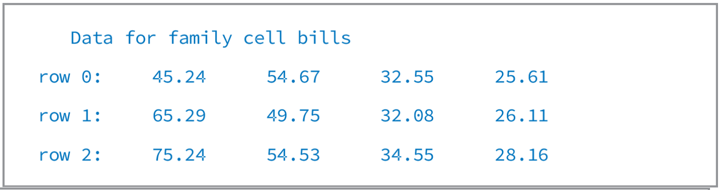
```
/* Working with a Two-Dimensional Array of Objects
   Anderson, Franceschi
*/

public class TwoDimAutoArray
{
 public static void main( String [ ] args )
 {
  // instantiate several Auto object references
  Auto sedan1 = new Auto( "BMW", 0, 0.0 );
  Auto sedan2 = new Auto( "BMW", 100, 15.0 );
  Auto sedan3 = new Auto( "Toyota", 0, 0.0 );
  Auto sportsCar = new Auto( "Ferrari", 0, 0.0 );
  Auto rv1 = new Auto( "Jeep", 0, 0.0 );
  Auto rv2 = new Auto( "Ford", 200, 30.0 );

  // declare and initialize two-dimensional array of Autos
  Auto [ ][ ] cars = { { sedan1, sedan2, sedan3 },
                       { sportsCar, new Auto( ) },
                       { rv1, rv2 } };

  Auto retrievedCar = cars[1][0];
  // retrievedCar gets the sportsCar object reference

  System.out.println( "cars[1][0]'s description is:\n"
                      + retrievedCar );
 }
}
```

Description

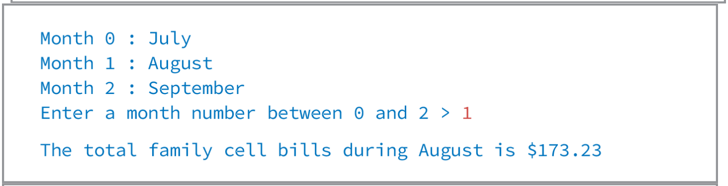
with these end-of-chapter questions

9.11.1	Multiple Choice Exercises Questions 1, 2, 3, 4, 5, 6, 7, 8
9.11.2	Reading and Understanding Code Questions 14, 15, 16, 17, 18
9.11.3	Fill In the Code Questions 33, 34
9.11.4	Identifying Errors in Code Questions 50, 51, 52, 53

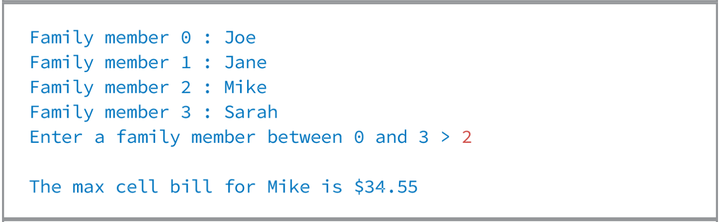
9.11.5	Debugging Area Question 59
9.11.6	Write a Short Program Question 65
9.11.8	Technical Writing Question 97



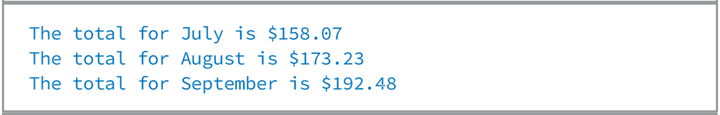
```
   Data for family cell bills
row 0:     45.24     54.67     32.55     25.61
row 1:     65.29     49.75     32.08     26.11
row 2:     75.24     54.53     34.55     28.16
```



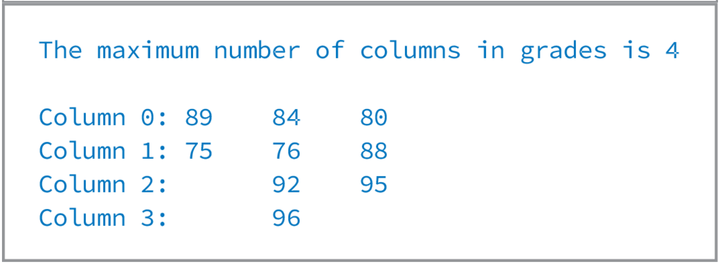
```
Month 0 : July
Month 1 : August
Month 2 : September
Enter a month number between 0 and 2 > 1
The total family cell bills during August is $173.23
```



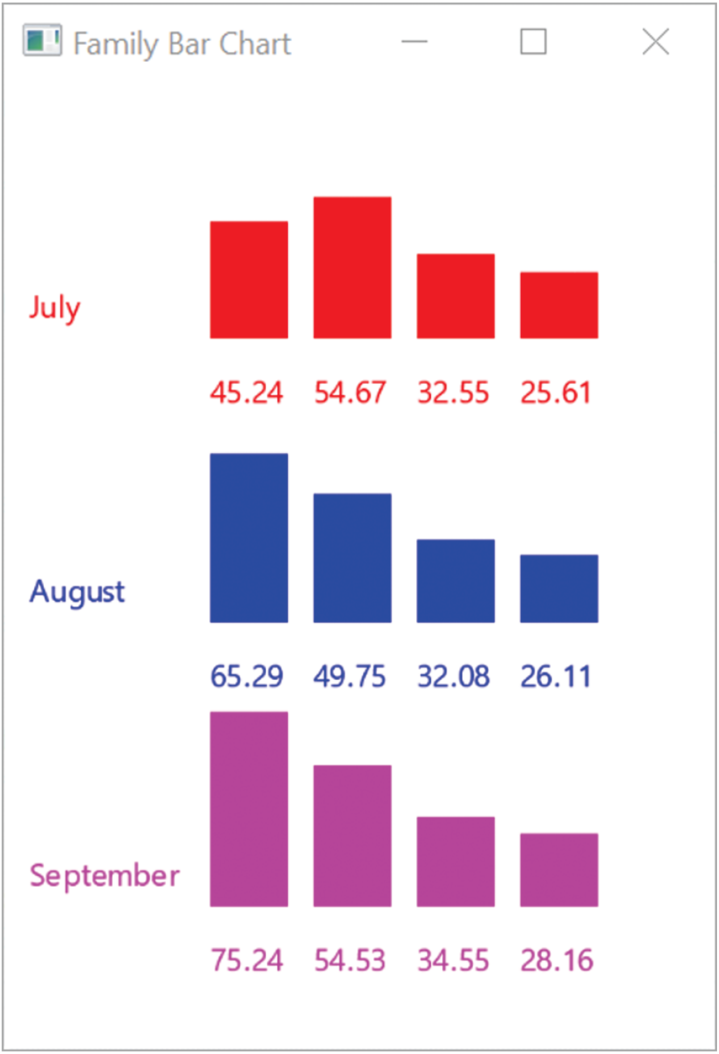
```
Family member 0 : Joe
Family member 1 : Jane
Family member 2 : Mike
Family member 3 : Sarah
Enter a family member between 0 and 3 > 2

The max cell bill for Mike is $34.55
```



```
The total for July is $158.07
The total for August is $173.23
The total for September is $192.48
```



```
The maximum number of columns in grades is 4

Column 0: 89    84    80
Column 1: 75    76    88
Column 2:       92    95
Column 3:       96
```


Family Bar Chart
July
45.24
54.67
32.55
25.61
August
65.29
49.75
32.08
26.11
September
75.24
54.53
34.55
28.16

9.3 Aggregate Two-Dimensional Array Operations

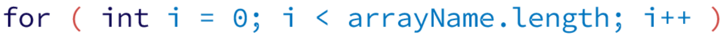
As with single-dimensional arrays, Java does not support aggregate operations on multidimensional arrays. For example, we cannot print the contents of an array using only the array name. Instead, we need to process each element individually.

9.3.1 Processing All the Elements of a Two-Dimensional Array

To process all the elements of a two-dimensional array, we use nested *for* loops that access and process each element individually. Often, the most logical way to process all elements is in row order, and within each row, in column order. We could also process elements one column at a time if that is more logical for the problem at hand.

In our nested *for* loops, the outer *for* loop will process the rows and the inner *for* loop will process the columns within each row. We will use *i* for the row index and *j* for the column index.

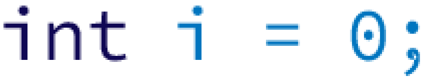
For the outer *for* loop, we can use the same header as we use to process single-dimensional arrays:



```
for ( int i = 0; i < arrayName.length; i++ )
```

Description

Note that the initialization statement of the outer loop:

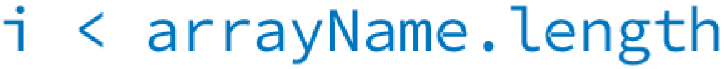


```
int i = 0;
```

Description

sets *i* to the index of the first row of the two-dimensional array. Then the outer loop update statement increments *i*, so that we process each row in order.

The outer loop condition:

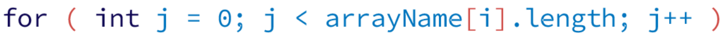


```
i < arrayName.length
```

Description

continues execution of the outer loop as long as the row index is less than the *length* of the two-dimensional array, which represents the number of rows. Note that we use the *less than* operator (<) instead of the *less than or equal to* operator (<=). Using the *less than or equal to* operator would cause us to illegally attempt to reference an element with a row index of *arrayName.length*.

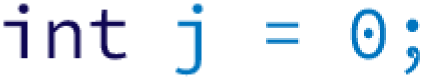
The *for* loop header for the inner loop, which processes the columns of the current row, is as follows:



```
for ( int j = 0; j < arrayName[i].length; j++ )
```

Description

The initialization statement of the inner loop:

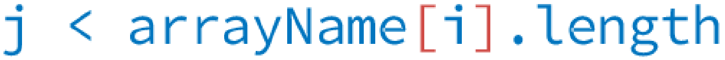


```
int j = 0;
```

Description

sets *j* to the index of the first column of the current row. Then the inner loop update statement increments *j* to the next column index, so that we process each column of the current row in order.

The inner loop condition:



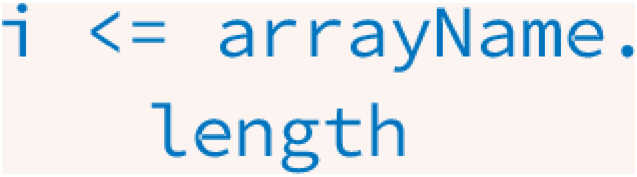
```
j < arrayName[i].length
```

Description

continues execution of the inner loop as long as the column index is less than the *length* of the current row (row *i*). Given that each row can have a different number of columns, this will ensure that we do not attempt to access an element beyond the last column index of the current row.

COMMON ERROR TRAP

In the outer *for* loop, using the following condition:

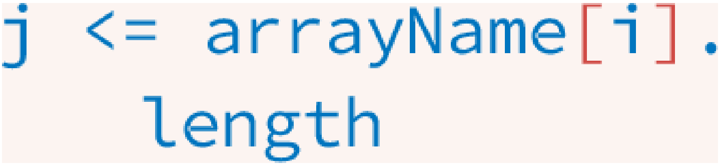


```
i <= arrayName.
    length
```

Description

will generate an *ArrayIndexOutOfBounds-Exception* because the last row of a two-dimensional array is *arrayName.length – 1*.

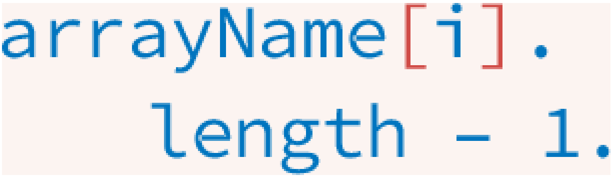
Similarly, in the inner *for* loop, using the condition:



```
j <= arrayName[i].
    length
```

Description

will generate an *ArrayIndexOutOfBounds-Exception* because the last column of row *i* in a two-dimensional array is

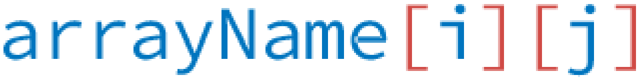


```
arrayName[i].
    length - 1.
```

Description

Note, again, that we use the *less than* operator (<), not the *less than or equal to* operator (<=), which would cause us to illegally attempt to reference an element with a column index of *arrayName[i].length*.

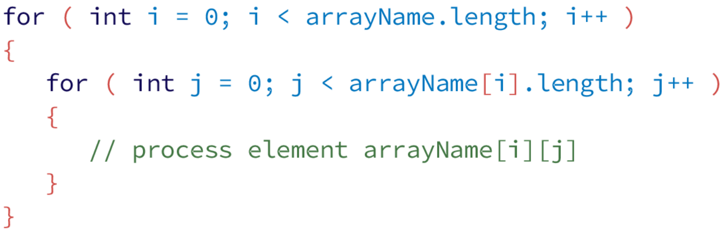
Inside the inner *for* loop, we refer to the current element being processed as:



```
arrayName[i][j]
```

Description

Thus, the general pattern for processing the elements of a two-dimensional array called *arrayName* in row-first, column-second order using nested *for* loops is:



```
for ( int i = 0; i < arrayName.length; i++ )
{
   for ( int j = 0; j < arrayName[i].length; j++ )
   {
      // process element arrayName[i][j]
   }
}
```

Description

Example 9.3 illustrates how to print all the elements of the two-dimensional array *familyCellBills* in row order. The array is declared and initialized at lines 10–12. At lines 16–23, the nested *for* loops, using the standard pattern described earlier, print all the elements of the array. Figure 9.6 shows the output of the program.

Figure 9.6 Output of Example 9.3

Description

Example 9.3 Two-Dimensional Array Processing

```java
/* Processing a Two-Dimensional Array of Cell Phone Bills
   Anderson, Franceschi
*/

public class OutputFamilyCellBills
{
 public static void main( String [ ] args )
 {
  // declare and initialize the array
  double [ ][ ] familyCellBills = { {45.24, 54.67, 32.55, 25.61},
                                    {65.29, 49.75, 32.08, 26.11},
                                    {75.24, 54.53, 34.55, 28.16} };

  System.out.println( "\tData for family cell bills" );

  for ( int i = 0; i < familyCellBills.length; i++ )
  {
    System.out.print( "\nrow " + i + ":\t" );
    for ( int j = 0; j < familyCellBills[i].length; j++ )
    {
      System.out.print( familyCellBills[i][j] + "\t" );
    }
  }
  System.out.println( );
 }
}
```

Description

9.3.2 Processing a Given Row of a Two-Dimensional Array

What if we want to process just one row of a two-dimensional array? For instance, we could be interested in calculating the sum of the cell bills for the whole family for a particular month, or identifying who had the highest cell bill in a particular month. The general pattern for processing the elements of row *i* of a two-dimensional array called *arrayName* uses a single *for* loop:

```java
for ( int j = 0; j < arrayName[i].length; j++ )
{
    // process element arrayName[i][j]
}
```

Description

Example 9.4 shows how to sum all the elements of a particular row of the two-dimensional array *familyCellBills*.

Example 9.4 Processing One Row in a Two-Dimensional Array

```
 1 /* Processing One Row of a Two-Dimensional Array
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6 import java.text.NumberFormat;
 7
 8 public class SumARowFamilyCellBills
 9 {
10  public static void main( String [ ] args )
11  {
12   // declare and initialize the array
13   double [ ][ ] familyCellBills = { {45.24, 54.67, 32.55, 25.61},
14                                     {65.29, 49.75, 32.08, 26.11},
15                                     {75.24, 54.53, 34.55, 28.16} };
16
17   String [ ] months = { "July", "August", "September" };
18   for ( int i = 0; i < months.length; i++ )
19       System.out.println( "Month " + i + " : " + months[i] );
20
21   Scanner scan = new Scanner( System.in );
22   int currentMonth;
23   do
24   {
25     System.out.print( "Enter a month number between 0 and 2 > " );
26     currentMonth = scan.nextInt( );
27   } while ( currentMonth < 0 || currentMonth > 2 );
28
29   double monthlyFamilyBills = 0.0;
30   for ( int j = 0; j < familyCellBills[currentMonth].length; j++ )
31   {
32     // add current family member bill to total
33     monthlyFamilyBills += familyCellBills[currentMonth][j];
34   }
35
36   NumberFormat priceFormat = NumberFormat.getCurrencyInstance( );
37   System.out.println( "\nThe total family cell bills during "
38                       + months[currentMonth] + " is "
39                       + priceFormat.format( monthlyFamilyBills ) );
40  }
41 }
```

Description

Since the rows correspond to the months, we declare and initialize at line 17 a single-dimensional *String* array named *months* in order to make our prompt more user-friendly. At lines

18–19, we print a menu for the user, providing month names and the corresponding indexes. At lines 23–27, we use a *do/while* loop to prompt the user for a month index until the user enters a valid value between 0 and 2.

To calculate the total of the family cell bills for the month index that the user inputs, we first initialize the variable *monthlyFamilyBills* to 0.0 at line 29. We then use a single *for* loop at lines 30–34, following the pattern described earlier, to sum all the family member bills for the month chosen by the user. We then format and output the total at lines 36–39. Figure 9.7 shows the output of the program when the user chooses 1 for the month.

Figure 9.7 Output of Example 9.4

Description

9.3.3 Processing a Given Column of a Two-Dimensional Array

If we want to determine the highest cell bill for Mike or calculate the average cell bill for Sarah, we will need to process just one column of the two-dimensional array.

The general pattern for processing the elements of column *j* of a two-dimensional array called *arrayName* uses a single *for* loop:

```
for ( int i = 0; i < arrayName.length; i++ )
{
    if ( j < arrayName[i].length )
       // process element arrayName[i][j]
}
```

Description

Because rows may have a different number of columns, a given row *i* may not have a column *j*. Thus, we need to check that the current column number is less than *arrayName[i].length* before we attempt to access *arrayName[i][j]*.

SOFTWARE ENGINEERING TIP

Before processing an element in a column, check whether the current row contains an element in that column. Doing so will avoid an *ArrayIndexOutOfBounds-Exception*.

Because our two-dimensional array *familyCellBills* has the same number of columns (4) in every row, no extra precaution is necessary here. It is a good software engineering practice, however, to verify that the column index is valid before attempting to process the array element.

Example 9.5 shows how to find the maximum value of all the elements of a particular column.

Example 9.5 Processing a Column in a Two-Dimensional Array

```
1 /* Processing One Column of a Two-Dimensional Array
2    Anderson, Franceschi
3 */
4
5 import java.util.Scanner;
6 import java.text.NumberFormat;
7
8 public class MaxMemberBill
9 {
10  public static void main( String [ ] args )
11  {
12   // declare and initialize the array
13   double [ ][ ] familyCellBills = { {45.24, 54.67, 32.55, 25.61},
14                                     {65.29, 49.75, 32.08, 26.11},
15                                     {75.24, 54.53, 34.55, 28.16} };
16
17   String [ ] familyMembers = { "Joe", "Jane", "Mike", "Sarah" };
18   for ( int i = 0; i < familyMembers.length; i++ )
19          System.out.println( "Family member " + i + " : "
20                              + familyMembers[i] );
21
22   Scanner scan = new Scanner( System.in );
23   int currentMember;
24   do
25   {
26    System.out.print( "Enter a family member between 0 and 3 > " );
27    currentMember = scan.nextInt( );
28   } while ( currentMember < 0 || currentMember > 3 );
29
30   double memberMaxBill = familyCellBills[0][currentMember];
31   for ( int i = 1; i < familyCellBills.length; i++ )
32   {
33    if ( currentMember < familyCellBills[i].length )
34    {
35      // update memberMaxBill if necessary
36      if ( familyCellBills[i][currentMember] > memberMaxBill )
37        memberMaxBill = familyCellBills[i][currentMember];
38    }
39   }
40
41   NumberFormat priceFormat = NumberFormat.getCurrencyInstance( );
42   System.out.println ( "\nThe max cell bill for "
43                        + familyMembers[currentMember] + " is "
44                        + priceFormat.format( memberMaxBill ) );
45  }
46 }
```

Description

At line 17, we declare and initialize a single-dimensional *String* array named *familyMembers* to make our prompt more user-friendly. At lines 24–28, we again use a *do/while* loop to prompt the user for a valid family member index.

To calculate the maximum value of the family member cell bills, we first initialize the variable *memberMaxBill* to the first element in the column (*familyCellBills[0][currentMember]*) at line

30. We then use a standard *for* loop at lines 31–39, following the pattern described earlier to update the value of *memberMaxBill* as necessary. There is one minor difference; we do not need to start the row at index 0 because we initialized *memberMaxBill* to the value of the element in row 0 of the column *currentMember*. Note that we assume that there is an element at column 0 of each row; that is, each row has been instantiated. The value of the variable *memberMaxBill* is then formatted and printed at lines 41–44. Figure 9.8 shows the output of the program.

Figure 9.8 Output of Example 9.5

Description

9.3.4 Processing a Two-Dimensional Array One Row at a Time

Earlier, we calculated the sum of the elements of a given row of a two-dimensional array. But what if we are interested in calculating that sum for each row? In this case, we need to initialize our total variable before we process each row and print the results after we process each row.

The general pattern for processing each row of a two-dimensional array called *arrayName* using nested *for* loops is

```
   for ( int i = 0; i < arrayName.length; i++ )
   {
      // initialize processing variables for row i
   for ( int j = 0; j < arrayName[i].length; j++ )
   {
       // process element arrayName[i][j]
   }
   // finish the processing of row i
}
```

Description

There are two important additions to the general pattern for processing all elements of the array:

- Before processing each row, that is, before the inner loop, we need to initialize the processing variables for the current row. If we are summing elements, we initialize the total variable to 0. If we are calculating a minimum or maximum value, we initialize the current minimum or maximum to the value of the first element of the current row.
- When we reach the end of each row, that is, after each completion of the inner loop, we finish processing the current row. For instance, we may want to print the sum or maximum value for that row.

Example 9.6 shows how to sum the elements of each row of the two-dimensional array *familyCellBills*.

Example 9.6 Processing Each Row in a Two-Dimensional Array

```
/* Processing Each Row of a Two-Dimensional Array
   Anderson, Franceschi
*/

import java.util.Scanner;
import java.text.NumberFormat;

public class SumEachRowFamilyCellBills
{
 public static void main( String [ ] args )
 {
  // declare and initialize the array
  double [ ][ ] familyCellBills = { {45.24, 54.67, 32.55, 25.61
                                    {65.29, 49.75, 32.08, 26.11
                                    {75.24, 54.53, 34.55, 28.16

  String [ ] months = { "July", "August", "September" };

  NumberFormat priceFormat = NumberFormat.getCurrencyInstance(
  double currentMonthTotal;
  for ( int i = 0; i < familyCellBills.length; i++ )
  {
   currentMonthTotal = 0.0;  // initialize total for row
   for ( int j = 0; j < familyCellBills[i].length; j++ )
   {
    // add current family member bill to current monthly total
    currentMonthTotal += familyCellBills[i][j];
   }
   // print total for row
   System.out.println( "The total for " + months[i] + " is "
                        + priceFormat.format( currentMonthTotal
  }
 }
}
```

Description

Again, the rows correspond to the months, and we declare and initialize at line 17 a *String* array named *months* in order to make the output user-friendly.

To calculate the total of the family cell bills for each month, we use nested *for* loops at lines 21–32, following the pattern described earlier.

Inside the outer *for* loop, we initialize the *currentMonthTotal* at line 23 before processing each row. Without this statement, the variable *currentMonthTotal* would continue to accumulate, as if we were summing all the elements of the array instead of calculating a separate sum for each row.

After the inner loop finishes, we complete the processing of row *i* by printing the value of *currentMonthTotal* at lines 29–31. **Figure 9.9** shows the output of the program.

Figure 9.9 Output of Example 9.6

Description

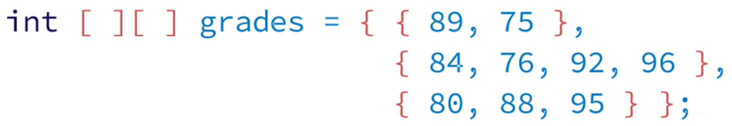
COMMON ERROR TRAP

Failing to initialize the row processing variables before each row is a logic error and will generate incorrect results.

9.3.5 Processing a Two-Dimensional Array One Column at a Time

Processing each column of a two-dimensional array requires a little extra checking. If the number of columns in each row differs, we must be careful not to attempt to access an element with an out-of-bounds column index. Generally, we will need to determine the number of columns in the largest row in the array before coding the outer loop header.

For example, suppose we are keeping track of our test grades in three classes: Intro to Java, Database Management, and English Composition. We have two test grades in Intro to Java, four in Database Management, and three in English Composition. We can use a two-dimensional array to store these test grades as follows:

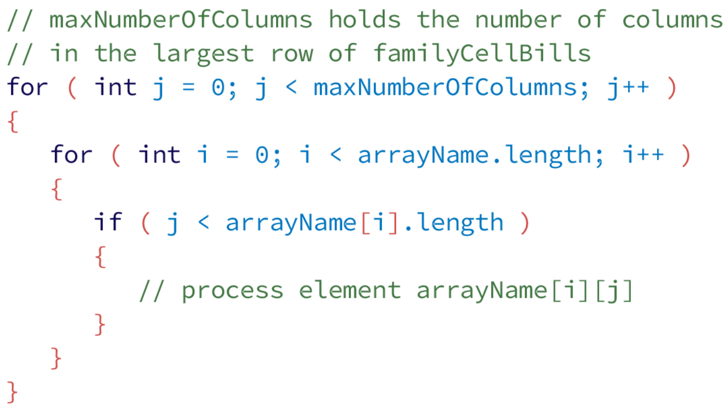


```
int [ ][ ] grades = { { 89, 75 },
                      { 84, 76, 92, 96 },
                      { 80, 88, 95 } };
```

Description

There are three rows in the array *grades*. The maximum number of columns in any row is four; therefore, in order to process all the columns, our outer loop should loop from column index 0 to column index 3. Our inner loop should check that the current column number exists in the row before attempting to process the element.

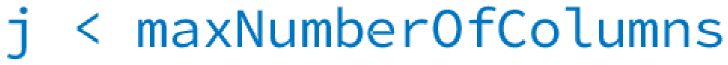
Let’s assume, at this point, that we stored the maximum number of columns in an *int* variable called *maxNumberOfColumns*. The general pattern for processing elements of a two-dimensional array, *arrayName*, one column at a time is:



```
// maxNumberOfColumns holds the number of columns
// in the largest row of familyCellBills
for ( int j = 0; j < maxNumberOfColumns; j++ )
{
   for ( int i = 0; i < arrayName.length; i++ )
   {
      if ( j < arrayName[i].length )
      {
         // process element arrayName[i][j]
      }
   }
}
```

Description

The outer loop condition:

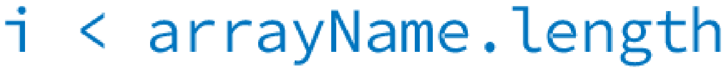


```
j < maxNumberOfColumns
```

Description

continues execution of the outer loop as long as the column index is less than the maximum number of columns of the two-dimensional array, which has been computed and assigned to the variable *maxNumberOfColumns*.

The inner loop condition:

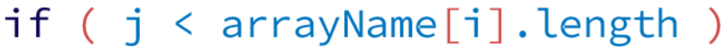


```
i < arrayName.length
```

Description

continues execution of the inner loop as long as the row index is less than the number of rows.

Again, because each row may have a different number of columns, a given row *i* may not have a column *j*. Thus, using the following *if* condition, we check that an element in column *j* exists—*j* is less than *arrayName[i].length*—before we attempt to access *arrayName[i][j]*:



```
if ( j < arrayName[i].length )
```

Description

Example 9.7 shows how this pattern can be implemented in a program.

Example 9.7 Processing a Two-Dimensional Array in Column Order

```
 1 /* Processing Each Column in a Two-Dimensional Array
 2    Anderson, Franceschi
 3 */
 4
 5 public class GradesProcessing
 6 {
 7  public static void main( String [ ] args )
 8  {
 9   int [ ][ ] grades = { { 89, 75 },
10                         { 84, 76, 92, 96 },
11                         { 80, 88, 95 } };
12
13   // compute the maximum number of columns
14   int maxNumberOfColumns = grades[0].length;
15   for ( int i = 1; i < grades.length; i++ )
16   {
17    if ( grades[i].length > maxNumberOfColumns )
18       maxNumberOfColumns = grades[i].length;
19   }
20   System.out.println( "The maximum number of columns in grades is "
21                       + maxNumberOfColumns );
22
23   for ( int j = 0; j < maxNumberOfColumns; j++ )
24   {
25    System.out.print( "\nColumn " + j + ": " );
26    for ( int i = 0; i < grades.length; i++ )
27    {
28     if ( j < grades[i].length )
29          System.out.print( grades[i][j] );
30     System.out.print( "\t" );
31    }
32   }
33   System.out.println( );
34  }
35 }
```

Description

The array *grades* is declared and initialized at lines 9–11. Lines 13–19 compute the maximum number of columns in a row and store the value in the *int* variable *maxNumberOfColumns*. First, we initialize *maxNumberOfColumns* to the number of columns of row 0 at line 14. At lines 15 to 19, we loop through each remaining row in *grades* and update *maxNumberOfColumns* if we find that the current row has more columns than *maxNumberOfColumns*.

At lines 23–32, we use nested loops to print all the elements of *grades* in column order, following the general pattern described earlier. The output of the program is shown in **Figure 9.10**.

Figure 9.10 The Output of Example 9.7

Description

9.3.6 Displaying Two-Dimensional Array Data as a Bar Chart

Another way to display two-dimensional array data is graphically, by drawing a bar chart. For example, the bar chart in **Figure 9.11** displays the data in the *familyCellBills* array.

Figure 9.11 The *familyCellBills* Array as a Bar Chart

Description

Each bar is a rectangle. In order to draw a rectangle, we need to provide four arguments to the *fillRect* method of the *GraphicsContext* class: the *x* and *y* coordinate of the upper-left corner of the rectangle and the width and height of the rectangle. We use the variables *xStart* and *yStartBar*, adjusted by the value of the array element, for the *x* and *y* coordinates of the rectangle. We use a constant, *BAR_WIDTH*, for the width of each rectangle, and the value of the array element for its height.

So to create a bar chart, we use our standard nested *for* loops, and call the *fillRect* method of the *GraphicsContext* class to draw a rectangle for each element. We use the *fillText* method to display the value of each element. To change colors for each row, we set up an array of *Color* objects, and loop through the array to set the current color for each row iteration. Furthermore, each time we process a row, we must reset the (*x*, *y*) coordinates of the first bar for the next row.

Example 9.8 shows the code that displays the bar chart shown in Figure 9.11.

Example 9.8 Displaying a Two-Dimensional Array as a Bar Chart

```
/* A two-dimensional bar chart application
   Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;

public class FamilyBarChartApplication extends Application
{
  final int LEFT_MARGIN = 80;         // starting x coordinate
  final int BASE_Y_BAR  = 100;        // bottom of the bars
  final int BASE_Y_VALUE = 125;       // bottom of the values
  final int BAR_WIDTH = 30;           // width of each bar
  final int SPACE_BETWEEN_BARS = 10;  // pixels between bars
  final int ROW_HEIGHT = 110;         // pixels between rows

  double [ ][ ] familyCellBills = { {45.24, 54.67, 32.55, 25.61},
                                    {65.29, 49.75, 32.08, 26.11},
                                    {75.24, 54.53, 34.55, 28.16} };

  String [ ] months = { "July", "August", "September" };
  Color [ ] colors = { Color.RED, Color.BLUE, Color.MAGENTA };

  @Override
  public void start( Stage stage )
  {
    // set up window title and size
    GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
                  stage, "Family Bar Chart", 275, 375 );

    int xStart = LEFT_MARGIN;   // x value for 1st column (bars)
    int yStart = BASE_Y_VALUE;  // y value for 1st row (data)
    int yStartBar = BASE_Y_BAR; // y value for 1st row (bars)

    for ( int i = 0; i < familyCellBills.length; i++ )
    {
      gc.setFill( colors[i] ); // set color for current row
      gc.fillText( months[i], xStart - LEFT_MARGIN + 10,
                              yStart - .3 * ROW_HEIGHT );

      for ( int j = 0; j < familyCellBills[i].length; j++ )
      {
        gc.fillRect( xStart, yStartBar - familyCellBills[i][j],
                     BAR_WIDTH, familyCellBills[i][j] );

        gc.fillText( Double.toString( familyCellBills[i][j] ),
                   xStart, yStart );

        // move to starting x value for next bar
        xStart += BAR_WIDTH + SPACE_BETWEEN_BARS;
      }

      // new row: increase yStart and yStartBar
      yStart += ROW_HEIGHT;
      yStartBar += ROW_HEIGHT;
      xStart = LEFT_MARGIN;
    }
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description

The *Color* single-dimensional array *colors* that we use to determine the color of each row of bars is declared and initialized at line 24. The first row of bars will be displayed in red, the second row in blue, and the third row in magenta. We use a *String* single-dimensional array, *months*, to label each row. Both arrays have the same number of rows as *familyCellBills*.

In the *start* method, at the beginning of the outer loop and before the inner loop, we set the color for the current row (line 39) by using the row number as an index into the *colors* array. At lines 40–41, we display the month.

In the body of the inner loop (lines 45–52), we draw the rectangle for the element value at row *i* and column *j* of *familyCellBills*, then display a *String* representing the same value. We then increment *xStart* to the location of the next bar to draw.

After the inner loop and before restarting the outer loop, we update the values of *yStart*, *yStartBar*, and *xStart* (lines 55–58) so that they are properly set for processing the next row. Earlier, we said that initializing variable values for the next row is usually done at the beginning of the outer loop body before entering the inner loop, but it also can be done after the inner loop and before re-entering the outer loop, as shown here.

CODE IN ACTION

To see a step-by-step illustration showing how to use two-dimensional arrays, look for the movie within the online resources. Click on the link to this chapter to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

with these end-of-chapter questions

9.11.2	Reading and Understanding Code Questions 19, 20, 21, 22, 23, 24, 25, 26, 27, 28
9.11.3	Fill In the Code Questions 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45
9.11.4	Identifying Errors in Code

	Question 54
9.11.5	Debugging Area Questions 60, 61, 62
9.11.6	Write a Short Program Questions 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79

```
Total votes per candidate
Smith      Jones      Berry      Chase
1543       2615       1845       2534

Total votes per district
1        2        3        4        5        6
873      1955     1156     896      2185     1472
```

9.4 Two-Dimensional Arrays Passed to and Returned from Methods

Writing methods that take two-dimensional arrays as parameters and/or return two-dimensional arrays is similar to working with single-dimensional arrays.

The syntax for a method that accepts a two-dimensional array as a parameter is the following:

```
returnType methodName( arrayType [ ][ ] arrayParameterName )
```

Description

The syntax for a method that returns a two-dimensional array is the following:

```
returnArrayType [ ][ ] methodName( parameterList )
```

Description

The caller of the method passes the argument list and assigns the return value to a reference to a two-dimensional array of the appropriate data type.

Combining both possibilities, the syntax for a method that accepts a two-dimensional array as a parameter and whose return value is a two-dimensional array is the following:

```
returnArrayType [ ][ ] methodName( arrayType [ ][ ] arrayParameterName )
```

Description

The caller of the method simply passes the name of the array without any brackets and assigns the return value to a reference to a two-dimensional array of the appropriate data type.

For example, suppose we want to tally votes in an election. We have four candidates running in six districts. We want to know how many votes each candidate received and how many votes were cast in each district. Thus, we can set up a two-dimensional array with each row representing a district and each column representing a candidate, with the values in each element representing the votes a candidate received in that district. We need to compute the sum of each row to find the number of votes per district and the sum of each column to find the number of votes per candidate.

To do this, we create a class, *Tally*, that has a two-dimensional array instance variable, *voteData*, storing the votes. The *Tally* class also has a method, *arrayTally*, that will compute the sums for each column and row of *voteData*. The sums will be returned from the method as a two-dimensional array with two rows. The first row will hold the totals for each column of *voteData*, and the second row will hold the totals for each row of *voteData*.

Example 9.9 shows the *Tally* class.

Example 9.9 The *Tally* Class

```
 1 /** Two-Dimensional Arrays as Method Parameters
 2 *   and Return Values: the Tally class
 3 *   Anderson, Franceschi
 4 */
 5
 6 public class Tally
 7 {
 8   int [ ][ ] voteData;
 9
10   /** constructor
11   * @param     voteData a 2-D array of vote counts
12   */
13   public Tally( int [ ][ ] voteData )
14   {
15     this.voteData = new int [voteData.length][ ];
16     for ( int i = 0; i < voteData.length; i++ )
17         this.voteData[i] = new int [voteData[i].length];
18
19      for ( int i = 0; i < voteData.length; i++ )
20      {
21        for ( int j = 0; j < voteData[i].length; j++ )
22        {
23          this.voteData[i][j] = voteData[i][j];
24        }
25      }
26   }
27
28   /** arrayTally method
29   *   @return a two-dimensional array of votes
30   */
31   public int [ ][ ] arrayTally( )
32   {
33     // create array of tallies, all elements are 0
34     int [ ][ ] returnTally = new int [2][ ];
35     returnTally[0] = new int [voteData[0].length];
36     returnTally[1] = new int [voteData.length];
37
38     for ( int i = 0; i < voteData.length; i++ )
39     {
40       for ( int j = 0; j < voteData[i].length; j++ )
41       {
42         returnTally[0][j] += voteData[i][j]; // add to column sum
43         returnTally[1][i] += voteData[i][j]; // add to row sum
44       }
45     }
46     return returnTally;
47   }
48 }
```

Description

The constructor, coded at lines 10–26, receives the two-dimensional array argument *voteData*. After instantiating the instance variable *voteData* at line 15, we copy the parameter *voteData* into the instance variable *voteData* one element at a time at lines 19–25.

We coded the *arrayTally* method at lines 28–47. Our first job is to instantiate the *returnArray*, which is the array the method will return to the caller. We know that the array will have two rows, one holding the sums of the columns and one holding the sums of the rows. Because each row in the *returnArray* will have a different number of columns, we instantiate the array with two rows, but do not give a value for the number of columns (line 34). We then instantiate each row with the appropriate number of columns (lines 35–36). Row 0, the sums of the columns, will have the same number of columns as the *voteData* array. In the interest of keeping this example simple, we have assumed that *voteData* has the same number of columns in every row, that is, each candidate was on the ballot in each district. Thus, that number is therefore equal to the number of columns in the first row, *voteData[0].length* (line 35). Row 1, the sum of the rows, will have the same number of columns as the number of rows in the *voteData* array.

In lines 38–45, we loop through the *voteData* array, computing the sums. We add each element's value to the sum for its column (line 42) and the sum for its row (line 43). When we finish, we return the *returnTally* array to the caller (line 46).

Example 9.10 shows a client program that instantiates a *Tally* object reference and calls the *arrayTally* method.

Example 9.10 The *VoteTally* Class

```
/** Tally votes: the VoteTally class
*   Anderson, Franceschi
*/

public class VoteTally
{
  public static void main( String [ ] args )
  {
    // votes are for 4 candidates in 6 districts.
    int [ ][ ] votes = { { 150, 253, 125, 345 },
                         { 250, 750, 234, 721 },
                         { 243, 600, 212, 101 },
                         { 234, 243, 143, 276 },
                         { 555, 343, 297, 990 },
                         { 111, 426, 834, 101 } };
    // candidate names
    String [ ] candidates = { "Smith", "Jones",
                              "Berry", "Chase" };

    // instantiate a Tally object reference
    Tally tally = new Tally( votes );

    // call arrayTally method to count the votes
    int [ ][ ] voteCounts = tally.arrayTally( );

    // print totals for candidates
    System.out.println( "Total votes per candidate" );
    for ( int i = 0; i < candidates.length; i++ )
      System.out.print( candidates[i] + "\t" );
    System.out.println( );
    for ( int j = 0; j < voteCounts[0].length; j++ )
      System.out.print( voteCounts[0][j] + "\t" );
    System.out.println( );

    // print totals for districts
    System.out.println("\nTotal votes per district" );
    for ( int i = 0; i < voteCounts[1].length; i++ )
       System.out.print( ( i + 1 ) + "\t\t" );
    System.out.println( );
    for ( int i = 0; i < voteCounts[1].length; i++ )
       System.out.print( voteCounts[1][i] + "\t" );
    System.out.println( );
  }
}
```

Description

We start by defining our two-dimensional array, *votes*, which holds the votes for each candidate for each district (lines 9–15). Most likely, we would read these values from a file, but for simplicity, we hard-coded the values in the initialization list. We also define a single-dimensional array of *Strings*, *candidates*, which holds the candidates' names (lines 16–18). Each name in the *candidates* array corresponds to the column in the *votes* array that holds that candidate's votes.

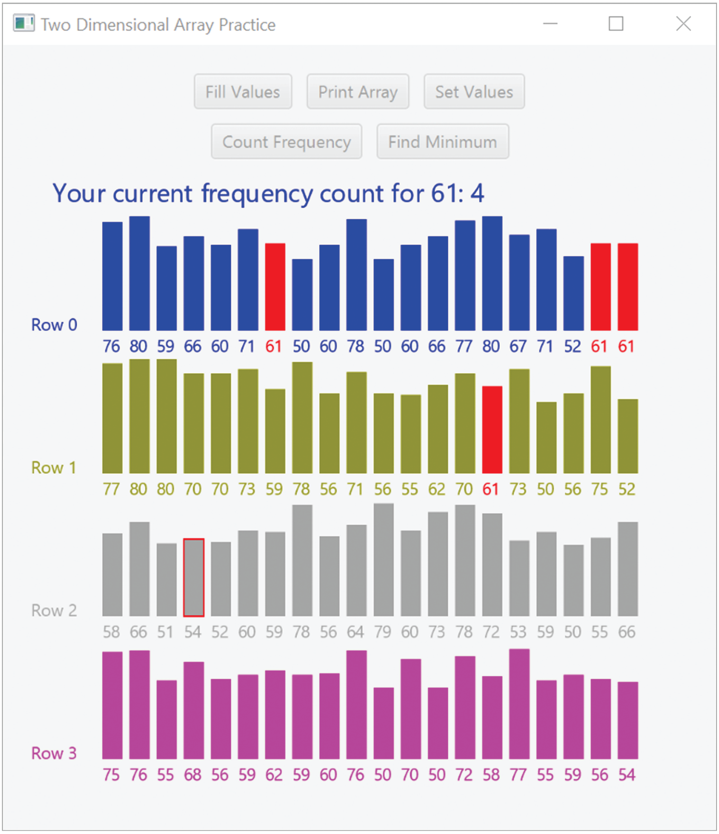
On lines 20–21, we instantiate the *Tally* object *tally*, passing the two-dimensional array *votes* to the *Tally* constructor. Notice that for the argument, we use only the array name, *votes*, without brackets.

On line 24, we call the *arrayTally* method, assigning the return value to a two-dimensional array reference named *voteCounts*.

Lines 26–33 print the totals per candidate by printing the elements in row 0 of the returned array, and lines 35–42 print the totals per district by printing the elements in row 1 of the returned array. The output is shown in Figure 9.12.

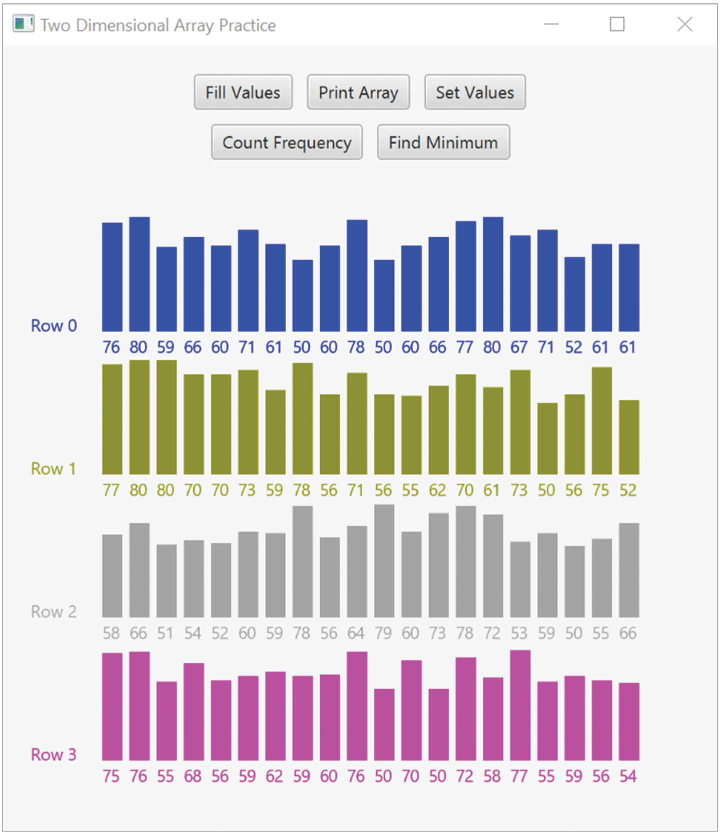
Figure 9.12 Output from Example 9.10

Description


Two Dimensional Array Practice
Fill Values
Print Array
Set Values
Count Frequency
Find Minimum

Your current frequency count for 61: 4

Row 0
76 80 59 66 60 71 61 50 60 78 50 60 66 77 80 67 71 52 61 61
Row 1
77 80 80 70 70 73 59 78 56 71 56 55 62 70 61 73 50 56 75 52
Row 2
58 66 51 54 52 60 59 78 56 64 79 60 73 78 72 53 59 50 55 66
Row 3
75 76 55 68 56 59 62 59 60 76 50 70 50 72 58 77 55 59 56 54


Two Dimensional Array Practice
Fill Values
Print Array
Set Values
Count Frequency
Find Minimum
Row 0
76 80 59 66 60 71 61 50 60 78 50 60 66 77 80 67 71 52 61 61
Row 1
77 80 80 70 70 73 59 78 56 71 56 55 62 70 61 73 50 56 75 52
Row 2
58 66 51 54 52 60 59 78 56 64 79 60 73 78 72 53 59 50 55 66
Row 3
75 76 55 68 56 59 62 59 60 76 50 70 50 72 58 77 55 59 56 54

9.5 Programming Activity 1: Working with Two-Dimensional Arrays

In this activity, you will work with a 4-row, 20-column, two-dimensional array of integers. Specifically, you will write methods to perform the following operations:

1. Fill the array with random numbers between 50 and 80.
2. Print the array.
3. Set every array element of a given row to a specified value. The value is a parameter of a method.
4. Find the minimum value in a given column of the array. The column is a parameter of a method.

5. Count the number of elements of the array having a specified value. The value is a parameter of a method.

The framework for this Programming Activity will animate your algorithm so that you can check the accuracy of your code. For example, Figure 9.13 shows the application counting the elements having the value 61.

Figure 9.13 Animation of the Programming Activity

Description

At this point, the application has found the value 61 in four array elements.

Instructions

In this chapter's Programming Activity 1 folder in the supplied code files, you will find the source files needed to complete this activity. Copy all the files to a folder on your computer. Note that all files should be in the same folder.

Open the *TwoDimArrayPracticeController.java* source file. Searching for five asterisks (*****) in the source code will position you at the sample method and the four other locations where you will add your code. We have provided the sample code for task number 1, which you can use as a model for completing the other tasks. In every task, you will fill in the code for a method that will manipulate an existing array of 4 rows and 20 columns. You should not instantiate the array; we have done that for you. Example 9.11 shows the section of the *TwoDimArrayPracticeController* source code where you will add your code.

Example 9.11 Location of Student Code in *TwoDimArrayPracticeController*

```
// ***** 1.  This method has been coded as an example
/** Fills the array with random numbers between 50 and 80
*  The instance variable named intArray is the integer array to be
*  filled with values
*/
public void fillValues( )
{
 Random rand = new Random( );
 for ( int row = 0; row < intArray.length; row++ )
 {
     System.out.print( row + "\t" );
     for ( int column = 0; j < intArray[row].length; column++ )
     {
      intArray[row][column] = rand.nextInt( 31 ) + 50;
      animate( row, column, -1 );  // needed for visual feedback
     }
     System.out.println( );
 }
}  // end of fillValues method

// ***** 2.  Student writes this method
/** Prints array to the console, elements are separated by a space
*  The instance variable named intArray is the integer array to be
*  printed
*/
public void printArray( )
{
  // Note:  To animate the algorithm, put this method call as the
  // last statement in your inner for loop
  //            animate( row, column, -1 );
  //      where row is the index of the array's current row
  //  and column is the index of the array's current column
  // Write your code here:

} // end of printArray method

// ***** 3.  Student writes this method
/** Sets all the elements in the specified row to the specified value
*  The instance variable named intArray is the integer array
*  @param value      the value to assign to the element of the row
*  @param row        the row in which to set the elements to value
*/
public void setValues( int value, int row )
{
   // Note:  To animate the algorithm, put this method call as the
   // last statement in your for loop
   //            animate( row, column, -1);
   //      where row is the index of the array's current row
   //      where column is the index of the array's current column
   // Write your code here:

}  // end of setValues method

// ***** 4.  Student writes this method
/** Finds minimum value in the specified column
*  The instance variable named intArray is the integer array
*  @param column     the column to search
*  @return           the minimum value found in the column
*/
public int findMinimum( int column )
{
  // Note:  To animate the algorithm, put this method call as the
  // last statement in your for loop
  //            animate( row, column, minimum );
  //  where row is the index of the array's current row
  //        column is the index of the array's current column
  //        minimum is the variable storing the current minimum
  // Write your code here:

  return 0; // replace this line with your return statement

}  // end of findMinimumn method

// ***** 5.  Student writes this method
/** Finds the number of times value is found in the array
*  The instance variable named intArray is the integer array
*  @param value     the value to count
*  @return          the number of times value was found
*/
public int countFound( int value )
{
   // Note:  To animate the algorithm, put this method call as the
   // last statement in your inner for loop
   //            animate( row, column, num );
   // where row is the index of the array's current row
   //       column is the index of the array's current column
   //       num is the local variable storing the current frequency
   //        count
   // Write your code here:

   return 0;  // replace this line with your return statement

}
// end of countFound method
```

Description

Note that for the *countFound* and *findMinimum* methods, we provide a dummy *return* statement: (*return 0;*). We do this so that the source code will compile. In this way, you can write and test each method separately, using step-wise refinement. When you are ready to write the *countFound* and *findMinimum* methods, just replace the dummy *return* statements with the appropriate *return* statement for that method.

The framework will animate your algorithm so that you can watch your code work. For this to happen, be sure that your single or nested *for* loops call the method *animate*. The arguments that you send to *animate* are not always the same and the location of the call to *animate* will differ depending on the task you are coding. Detailed instructions for each task are included in the code.

To test your code, compile *TwoDimArrayPracticeController.java* and run the *TwoDimArrayPracticeApplication* source code. Figure 9.14 shows the graphics window when the program begins. Because the values of the array are randomly generated, the values will be different each time the program runs. To test any method, click on the appropriate button.

Figure 9.14 The Graphics Window When the Application Begins

Description

Troubleshooting

If your method implementation does not animate, check these tips:

- Verify that the last statement in your single *for* loop or inner *for* loop is a call to the *animate* method and that you passed the appropriate arguments. For example:

  ```
  animate( row, column );
  ```

  Description

- Verify that your exit conditions for your *for* loops are correct. Sometimes the exit condition depends on the length of the array (i.e., the number of rows in the array), and sometimes it depends on the number of columns in the current row of the array.

DISCUSSION QUESTIONS

1. With a two-dimensional array, for which operations would you use nested *for* loops and for which operations would you use a single *for* loop?
2. When performing an operation on a given row, which index is fixed and which index is used as the looping variable? When performing an operation on a given column, which index is fixed and which index is used as the looping variable?

9.6 Other Multidimensional Arrays

Sometimes we might need an array with more than two dimensions. For example, we might be interested in keeping track of sales on a per-year, per-week, and per-day basis. In this case, we would use a three-dimensional array as follows:

1st dimension: year
2nd dimension: week
3rd dimension: day of the week

Earlier in this chapter, we explained that a two-dimensional array is an array of single-dimensional arrays. Similarly, a three-dimensional array is an array of two-dimensional arrays. And a four-dimensional array is an array of three-dimensional arrays. More generally, an *n*-dimensional array is an array of (*n* – 1)-dimensional arrays.

Table 9.3 shows how an *n*-dimensional array is structured dimension by dimension; i_1, i_2, ..., i_n are used as generic indexes for the first dimension, second dimension, ..., and nth dimensions.

Structure of an *n*-Dimensional Array

Dimension	**Array Element**
first	arrayName[i$_1$] is an (n – 1)-dimensional array
second	arrayName[i$_1$][i$_2$] is an (n – 2)-dimensional array
*k*th	arrayName[i$_1$][i$_2$][i$_3$][..][i$_k$] is an (n – k) multi-dimensional array

$(n-2)$th	arrayName[i_1][i_2][i_3][..][i_{n-2}] is a two-dimensional array
$(n-1)$th	arrayName[i_1][i_2][i_3][..][i_{n-1}] is a single-dimensional array
nth	arrayName[i_1][i_2][i_3][..][i_{n-1}][i_n] is an array element

Description

If we keep track of sales over a period of 10 years, then we would have a 10-by-52-by-7 array. The principles discussed for a two-dimensional array still apply; we just have three dimensions instead of two. The following code sequence illustrates how to declare, instantiate, and access elements of this three-dimensional array:

```
double [ ][ ][ ] Sales;            // declare a four-dimensional array
Sales = new double [10][52][7];    // instantiate the array
Sales[0][0][0] = 638.50;           // access the first element
sales[4][22][3] = 928.30;          // access another element
sales[9][51][6] = 1234.90;         // access the last element
```

Description

To process elements of a single-dimensional array, we use a simple *for* loop; for a two-dimensional array, we use a double *for* loop. For a three-dimensional array, we use a triple *for* loop.

The general pattern for processing elements in a three-dimensional array is

```
for ( int i = 0; i < arrayName.length; i++ )
{
    for ( int j = 0; j < arrayName[i].length; j++ )
    {
       for ( int k = 0; k < arrayName[i][j].length; k++ )
       {
          // access and process the element arrayName[i][j][k]
       }
    }
}
```

Description

The following code sequence will print the elements of the three-dimensional array *sales*:

```
for ( int i = 0; i < sales.length; i++ )
{
   for ( int j = 0; j < sales[i].length; j++ )
   {
      for ( int k = 0; k < sales[i][j].length; k++ )
      {
       // access the element at sales[i][j][k]
       System.out.println( sales[i][j][k] + "\t" );
      }
       // skip a line when second dimension index changes
       System.out.println( );
   }
   // skip a line when first dimension index changes
   System.out.println( );
}
```

Description

If we are interested in keeping track of sales on a state-by-state basis, we can use a four-dimensional array as follows:

1st dimension: state
2nd dimension: year
3rd dimension: week
4th dimension: day of the week

The following code sequence illustrates how to declare, instantiate, and access the elements of such a four-dimensional array:

```
double [ ][ ][ ][ ] stateSales;           // declare a four-dimensional
                                          // array
stateSales = new double [50][10][52][7];  // instantiate the array
stateSales[0][0][0][0] = 58.50;           // access the first element
sales[34][4][22][3] = 98.30;              // access another element
sales[49][9][51][6] = 137.70;             // access the last element
```

Description

To process elements of a four-dimensional array, we use a quadruple *for* loop. That quadruple *for* loop pattern parallels the ones for the two-dimensional and three-dimensional arrays. For a four-dimensional array called *arrayName*, it is:

```
for ( int i = 0; i < arrayName.length; i++ )
{
    for ( int j = 0; j < arrayName[i].length; j++ )
    {
       for ( int k = 0; k < arrayName[i][j].length; k++ )
       {
          for ( int l = 0; l < arrayName[i][j][k].length; l++ )
          {
             // process element arrayName[i][j][k][l]
          }
       }
    }
}
```

Description

```
Using the traditional for loop:
34     89     65

Using the enhanced for loop:
34     89     65

Using unboxing and enhanced for loop:
34     89     65

After calling set( 1, 100 ):
34     100      65

At index 0, 34 was removed

After removing the element at index 0:
100     65
```

```
Enter a keyword > Java
Our book collection is:
title: Intro to Java     author: James     price: 56.99
title: Advanced Java     author: Green     price: 65.99
title: Java Servlets     author: Brown     price: 75.99
title: Intro to HTML     author: James     price: 29.49
title: Intro to Flash    author: James     price: 34.99
title: Advanced HTML     author: Green     price: 56.99

Enter a keyword > Java
The search results for Java are:
title: Intro to Java     author: James     price: 56.99
title: Advanced Java     author: Green     price: 65.99
title: Java Servlets     author: Brown     price: 75.99
```

9.7 The *ArrayList* Class

As we have seen, single-dimensional and multidimensional arrays are useful in many situations. However, they have limitations.

Let's say we are designing a search engine for a large website, for example, an online bookstore. The user will type a word in a text field box, our code will access a database, retrieve all the books with titles that contain this word, and return them to the user.

We could store the book information in an array of books. One problem, however, is that we don't know how many books we will have. There could be 3, 32, 500, or 5,000 books, or maybe even more. Without knowing the number of books, we do not know what size to make the array. The safest bet would be to create the array with the maximum possible number of elements, that is, the maximum number of books that we anticipate. If we actually have fewer books than we anticipated, however, we will waste space.

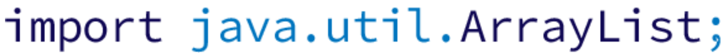
And if we end up with more books than we anticipated, we would need to increase the size of the array. Because the size of an array is fixed when it is instantiated, changing the size of an array is a tedious process. We have to instantiate a new array and copy the elements of the original array to the new array.

The *ArrayList* class, in the *java.util* package, solves these problems. An *ArrayList* object automatically expands its capacity as needed. The *ArrayList* class uses **generics**. Generics are **parameterized types**, meaning that the data type will be defined at the time a client class declares and instantiates an object of the class. Generics allow programmers to design and code classes that use objects without specifying the class—or data type—of the object.

Thus, for example, we could have an *ArrayList* of *Book* objects, an *ArrayList* of *Auto* objects, or an *ArrayList* of *Strings*. The specified type must be a class, not a primitive type. If we want to store primitive data types in an *ArrayList*, then we need to use one of the wrapper classes such as *Integer*, *Double*, or *Character*.

The *ArrayList* class, and more generally a class using generics, can be used for many purposes. This is another facet of object-oriented programming that allows programmers to reuse code.

Because the *ArrayList* class is in the *java.util* package, programs using an *ArrayList* object will need to provide the following *import* statement:

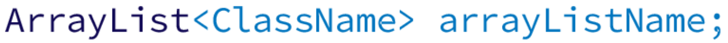


```
import java.util.ArrayList;
```

Description

9.7.1 Declaring and Instantiating *ArrayList* Objects

Here is the syntax for declaring an *ArrayList* of objects:

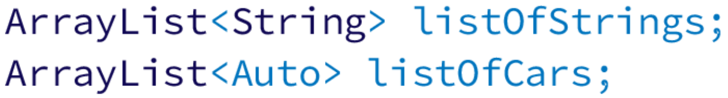


```
ArrayList<ClassName> arrayListName;
```

Description

Inside the brackets, we declare the class type of the objects that will be stored in the *ArrayList*. A space is optional between the *ArrayList* class name and the opening bracket.

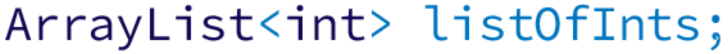
For example, these two statements declare an *ArrayList* of *Strings* and an *ArrayList* of *Auto* objects:



```
ArrayList<String> listOfStrings;
ArrayList<Auto> listOfCars;
```

Description

If we try to declare an *ArrayList* object reference using a primitive data type instead of a class type, as in



```
ArrayList<int> listOfInts;
```

Description

we will get this compiler error:

```
Test.java:7: error: unexpected type
  ArrayList<int> listOfInts;
            ^
required: reference
found:    int
1 error
```

Description

Two constructors of the *ArrayList* class are shown in Table 9.4.

ArrayList Constructors

ArrayList Constructors Summary

Constructor name and argument list
`ArrayList<ClassName>( )` constructs an *ArrayList* object of *ClassName* type with an initial capacity of 10
`ArrayList<ClassName>( int initialCapacity )` constructs an *ArrayList* object of *ClassName* type with the specified initial capacity

Description

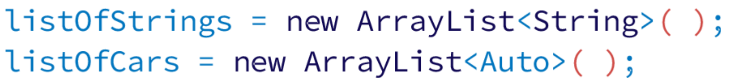
If we know how many elements we will store in the *ArrayList* object, we can use the overloaded constructor to specify the initial capacity; otherwise, we can simply use the default constructor. As we add elements to the *ArrayList* object, its capacity will increase automatically, as needed.

Here is the syntax for instantiating an *ArrayList* using the default constructor:

```
arrayListName = new ArrayList<ClassName>( );
  where ClassName is the class type of the objects that will be stored
  in the ArrayList and arrayListName has been declared previously as an
  ArrayList reference for that class.
```

Description

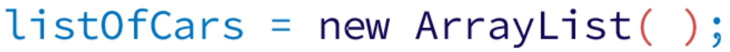
These statements will instantiate the *ArrayList* objects declared earlier, with an initial capacity of 10:



```
listOfStrings = new ArrayList<String>( );
listOfCars = new ArrayList<Auto>( );
```

Description

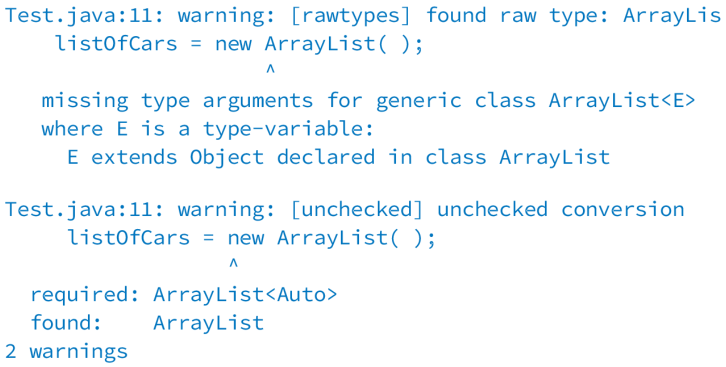
If we try to instantiate an *ArrayList* object without specifying the object type, as in



```
listOfCars = new ArrayList( );
```

Description

we get the following warnings from the compiler (using Xlint):



```
Test.java:11: warning: [rawtypes] found raw type: ArrayLis
    listOfCars = new ArrayList( );
                                 ^
   missing type arguments for generic class ArrayList<E>
   where E is a type-variable:
     E extends Object declared in class ArrayList

Test.java:11: warning: [unchecked] unchecked conversion
     listOfCars = new ArrayList( );
                              ^
  required: ArrayList<Auto>
  found:    ArrayList
2 warnings
```

Description

In *ArrayLists*, there is a distinction between capacity and size. The **capacity** of an *ArrayList* is the number of elements allocated to the list. The **size** is the number of those elements that are filled with objects. Thus, when we instantiate an *ArrayList* using the default constructor, its capacity is 10, but its size is 0. In other words, the *ArrayList* has room for 10 objects, but no objects are currently stored in the list.

These statements will declare, then instantiate, an *ArrayList* of *String* objects with an initial capacity of 5, using the overloaded constructor:

```
ArrayList<String> listOfStrings1;
listOfStrings1 = new ArrayList<String>( 5 );
```

Description

In this case, the capacity of *listOf Strings1* is 5 and its size is 0.

We can also combine the declaration and instantiation of an *ArrayList* object into one statement. Here is the syntax using the default constructor:

```
ArrayList<ClassName> arrayListName = new ArrayList<ClassName>( );
```

Description

These statements will declare and instantiate two *ArrayList* objects of *Integers* and *Strings*, respectively:

```
ArrayList<Integer> listOfInts = new ArrayList<Integer>( );
ArrayList<String> listOfStrings2 = new ArrayList<String>( );
```

Description

9.7.2 Methods of the *ArrayList* Class

Like arrays, the *ArrayList* class uses indexes to refer to elements. Among others, it provides methods that provide the following functions:

- add an item at the end of the list
- replace an item at a given index
- remove an item at a given index
- remove all the items in the list
- search the list for a specific item
- retrieve an item at a given index
- retrieve the index of a given item
- check to see if the list is empty

- return the number of items in the list, that is, its size
- optimize the capacity of the list by setting its capacity to the number of items in the list

Some of the most useful methods are shown in Table 9.5. Note that some of the method headers include *E* as their return type or parameter data type (as opposed to a class name or simply the *Object* class). *E* represents the data type of the *ArrayList*. Thus, for an *ArrayList* of *Integer* objects, *E* is an *Integer*; and the *get* method, for example, returns an *Integer* object. Similarly, for an *ArrayList* of *Auto* objects, *E* is an *Auto* object. In this case, the *get* method returns an *Auto* object.

ArrayList Methods

Useful Methods of the *ArrayList* Class

Return value	Method name and argument list
`boolean`	`add( E element )` appends the specified *element* to the end of the list
`E`	`remove( int index )` removes and returns the element at the specified *index* position in the list
`void`	`clear( )` removes all the elements from this list
`E`	`get( int index )` returns the element at the specified *index* position in the list; the element is not removed from the list
`E`	`set( int index, E element )` replaces the element at the specified *index* position in this list with the specified *element*
`int`	`size( )`

	returns the number of elements in this list
`void`	`trimToSize( )` sets the capacity to the list's current size

Description

9.7.3 Looping Through an *ArrayList* Using an Enhanced *for* Loop

The general pattern for processing elements of an *ArrayList* of *ClassName* objects called *arrayListName* using a *for* loop is

```
ClassName currentObject;
for ( int i = 0; i < arrayListName.size( ); i++ )
{
      currentObject = arrayListName.get( i );
      // process currentObject
}
```

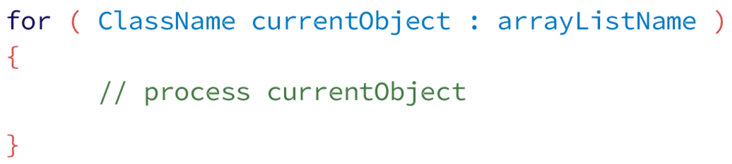
Description

For instance, to process elements of an *ArrayList* of *String* object references called *listOfStrings* using a standard *for* loop, the general pattern is:

```
String currentString;
for ( int i = 0; i < listOfStrings.size( ); i++ )
{
      currentString = listOfStrings.get( i );
      // process currentString
}
```

Description

Java provides a simplified way to process the elements of an *ArrayList*, called the **enhanced *for* loop.** The general pattern for processing elements of an *ArrayList* of *ClassName* objects called *arrayListName* using the enhanced *for* loop is:

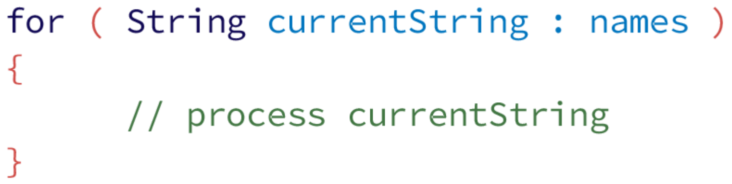


```
for ( ClassName currentObject : arrayListName )
{
      // process currentObject

}
```

Description

A variable of the class type of the objects stored in the *ArrayList* is declared in the enhanced *for* loop header, followed by a colon and name of the *ArrayList*. The enhanced *for* loop enables looping through the *ArrayList* objects automatically. Your code does not call the *get* method; inside the body of the loop, *currentObject* is directly available for processing.

For example, to process elements of an *ArrayList* of *Strings* called *names* using the enhanced *for* loop, the general pattern is:



```
for ( String currentString : names )
{
      // process currentString
}
```

Description

Example 9.12 shows how to create and use an *ArrayList* of *Integers*. Line 11 declares and instantiates the *ArrayList* object reference *list* using the default constructor. Three elements are added to *list* using the *add* method at lines 12–14. As the arguments to the *add* method, we use *ints.* The autoboxing feature of Java automatically converts an *int* to an *Integer* object when an *int* variable is used where an *Integer* object is expected.

Example 9.12 Using *ArrayList* Methods

```
 1 /* A Simple ArrayList of Integers
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.ArrayList;
 6
 7 public class ArrayListOfIntegers
 8 {
 9  public static void main( String [ ] args )
10  {
11   ArrayList<Integer> list = new ArrayList<Integer>( );
12   list.add( 34 ); // autoboxing
13   list.add( 89 ); // autoboxing
14   list.add( 65 ); // autoboxing
15
16   System.out.println( "Using the traditional for loop:" );
17   for ( int i = 0; i < list.size( ); i++ )
18      System.out.print( list.get( i ) + "\t" );
19   System.out.println( );
20
21   System.out.println( "\nUsing the enhanced for loop:" );
22   for ( Integer currentInteger : list )
23      System.out.print( currentInteger + "\t" );
24   System.out.println( );
25
26   System.out.println( "\nUsing unboxing and enhanced for loop:" );
27   for ( int currentInt : list ) // unboxing
28      System.out.print( currentInt + "\t" );
29   System.out.println( );
30
31   list.set( 1, 100 );
32   System.out.println( "\nAfter calling set( 1, 100 ):" );
33   for ( int currentInt : list ) // unboxing
34      System.out.print( currentInt + "\t" );
35   System.out.println( );
36
37   int removed = list.remove( 0 );
38   System.out.println( "\nAt index 0, " + removed + " was removed" );
39   System.out.println( "\nAfter removing the element at index 0:" );
40   for ( int currentInt : list ) // unboxing
41      System.out.print( currentInt + "\t" );
42   System.out.println( );
43  }
44 }
```

Description

After an *ArrayList* object has been declared and instantiated as being of a certain class type, you cannot add an object of a different class type. For example, you could not add a *Double* value to an *ArrayList* of *Integers*.

At lines 17–18, we print the elements of *list* using a traditional *for* loop, using the *get* method to retrieve the element at the current index. At lines 22–23, we use the enhanced *for* loop to print the elements. At lines 27–28, we also use the enhanced *for* loop to print the elements; but this time, we use an *int* as the looping variable, using the unboxing feature of Java, which converts *Integer* objects to *int* values, as needed. At line 31, we use the *set* method to change the value of the element at index 1 to 100, also using autoboxing. At line 37, we use the *remove* method to delete the element at index 0 and assign it to the variable *removed*, using unboxing again.

The output of this example is shown in **Figure 9.15**.

Figure 9.15 Output of Example 9.12

Description

9.7.4 Using the *ArrayList* Class in a Program

Now let's see how we can use the *ArrayList* class in a Java program. Going back to our example of a bookstore and a search engine, we want to design and code a simple program that enables users to search for books.

We will have three classes in this program:

- a *Book* class, encapsulating the concept of a book
- a *BookStore* class, encapsulating the concept of a bookstore
- a *BookSearchEngine* class, including the *main* method, which provides the user interface

In the interest of keeping things simple, our *Book* class will contain only three instance variables: the book title, which is a *String*; the book's author, which is also a *String*; and the book price, which is a *double*.

Example 9.13 shows a simplified *Book* class with constructors, accessor methods, and a *toString* method.

Example 9.13 The *Book* Class

```java
/* Book class
   Anderson, Franceschi
*/

public class Book
{
 private String title;
 private String author;
 private double price;

 /** default constructor
 */
 public Book( )
 {
  title = "";
  author = "";
  price = 0.0;
 }

 /** overloaded constructor
 * @param title    the value to assign to title
 * @param author   the value to assign to author
 * @param price    the value to assign to price
 */
 public Book( String title, String author, double price
 {
  this.title = title;
  this.author = author;
  this.price = price;
 }

 /** getTitle method
 *   @return the title
 */
 public String getTitle( )
 {
  return title;
 }

 /** getAuthor method
 *   @return the author
 */
 public String getAuthor( )
 {
  return author;
 }

 /** getPrice method
 *   @return the price
 */
 public double getPrice( )
 {
  return price;
 }

 /** toString
 * @return title, author, and price
 */
 @Override
 public String toString( )
 {
  return ( "title: " + title + "\t"
           + "author: " + author + "\t"
           + "price: " + price );
 }
}
```

Description

Our *BookStore* class, shown in **Example 9.14**, will simply have one instance variable: an *ArrayList* of *Book* objects, representing the collection of books in the bookstore, which we name *library*.

Example 9.14 The *BookStore* Class

```java
/*  BookStore class
    Anderson, Franceschi
*/

import java.util.ArrayList;

public class BookStore
{
 private ArrayList<Book> library;

 /** default constructor
 *   instantiates ArrayList of Books
 */
 public BookStore( )
 {
  library = new ArrayList<Book>( );
  library.add( new Book( "Intro to Java", "James", 56.99 ) );
  library.add( new Book( "Advanced Java", "Green", 65.99 ) );
  library.add( new Book( "Java Servlets", "Brown", 75.99 ) );
  library.add( new Book( "Intro to HTML", "James", 29.49 ) );
  library.add( new Book( "Intro to Flash", "James", 34.99 ) );
  library.add( new Book( "Advanced HTML", "Green", 56.99 ) );
  library.trimToSize( );
 }

 /** toString
 *  @return  each book in library, one per line
 */
 @Override
 public String toString( )
 {
  String result = "";
  for ( Book tempBook : library )
  {
    result += tempBook.toString( ) + "\n";
  }
  return result;
 }

 /** Generates list of books containing searchString
 * @param searchString       the keyword to search for
 * @return       the ArrayList of books containing the keyword
 */
 public ArrayList<Book> searchForTitle( String searchString )
 {
  ArrayList<Book> searchResult = new ArrayList<Book>( );
  for ( Book currentBook : library )
  {
   if ( currentBook.getTitle( ).indexOf( searchString ) != -1 )
        searchResult.add( currentBook );
  }
  searchResult.trimToSize( );
  return searchResult;
 }
}
```

Description

In most cases, when an *ArrayList* is filled with data, that data will come from a database or a file. In the interest of focusing on the *ArrayList* class and its methods, we have hard-coded the objects for the *ArrayList library* in the *BookStore* class, rather than reading them from a database or a file.

In the default constructor (lines 11 to 24), we instantiate the *library* instance variable, then add six *Book* objects to *library* using the *add* method from the *ArrayList* class. At line 23, we call the *trimToSize* method to set the capacity of *library* to its current size, which is 6, in order to minimize the memory resources used.

The *toString* method is coded from lines 26 to 38. It generates and returns a *String* representing all the books in *library*, one book per line. In order to do that, we use an enhanced *for* loop from lines 33 to 36. The header of that loop, at line 33, follows the general pattern of the enhanced *for* loop header by declaring a *Book* variable named *tempBook*, followed by a colon, followed by *library*, the *ArrayList* object to loop through.

The *searchForTitle* method, coded from lines 40 to 54, performs the task of searching for a keyword within the title of each *Book* object stored in *library*. The keyword, a *String*, is the parameter of the method and is named *searchString*. This method returns an *ArrayList* containing the *Book* objects that have the keyword in their title. We create that *ArrayList* of *Books*, which we name *searchResult* at line 46 and loop through *library* using an enhanced *for* loop from lines 47 to 51. Inside the body of the loop, we use the *indexOf* method of the *String* class to test if the current *Book* object contains the keyword *searchString* in its *title*. If it does, we add that *Book* object to *searchResult*. Finally, we call the method *trimToSize* to set the capacity of *searchResult* to the current number of elements, then return the *ArrayList* to the caller.

Our *BookSearchEngine* class, shown in Example 9.15, contains the *main* method: it creates a *BookStore* object, asks the user for a keyword, and searches for partial matches in our *BookStore* object.

Example 9.15 A Search Engine for Books

```
/* BookSearchEngine class
   Anderson, Franceschi
*/
```

```
 1 /* BookSearchEngine class
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.*;
 6
 7 public class BookSearchEngine
 8 {
 9  public static void main( String [ ] args )
10  {
11   BookStore bs = new BookStore( );
12
13   System.out.println( "Our book collection is:" );
14   System.out.println( bs.toString( ) );
15   Scanner scan = new Scanner( System.in );
16   System.out.print( "Enter a keyword > " );
17   String keyword = scan.next( );
18
19   ArrayList<Book> results = bs.searchForTitle( keyword );
20
21   System.out.println( "The search results for " + keyword
22                       + " are:" );
23   for ( Book tempBook : results )
24         System.out.println( tempBook.toString( ) );
25  }
26 }
```

Description

A *BookStore* object, *bs*, is declared and instantiated at line 11. Lines 13 and 14 simply output the collection of *Books* in the *BookStore* object *bs*; later, when the search results are output, we can compare that output to the original list of *Books* to check our results. At lines 15–17, the user is then prompted for a keyword that will be used to search for books whose title contains that keyword. At line 19, we call the *searchForTitle* method with *keyword* as its argument; the *ArrayList* of *Book* objects returned is assigned to the variable *results*. At lines 23–24, we loop through *results* and output its contents, again using the enhanced *for* loop. Figure 9.16 shows a run of the program with the user searching for books containing the word “Java.”

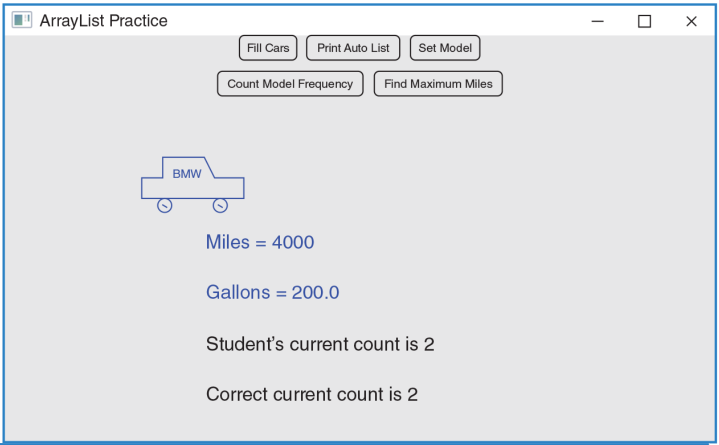
Figure 9.16 Results of a Search for the Keyword “Java”

Description

CODE IN ACTION

To see a step-by-step illustration showing how to use the *ArrayList* class, look for the movie within the online resources. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock


ArrayList Practice
Fill Cars
Print Auto List
Set Model
Count Model Frequency
Find Maximum Miles
BMW

Miles = 4000

Gallons = 200.0

Student’s current count is 2

Correct current count is 2

ArrayList Practice
Fill Cars
Print Auto List
Set Model
Count Model Frequency
Find Maximum Miles

9.8 Programming Activity 2: Working with the *ArrayList* Class

In this activity, you will work with an *ArrayList* object. Specifically, you will write the code to perform the following operations:

1. Fill the *ArrayList* object with *Auto* elements.
2. Print the *Auto* elements contained in the *ArrayList* object.
3. Set the *model* instance variable of every *Auto* element in the *ArrayList* object to a specified model.

4. Find the maximum number of miles of all *Auto* elements contained in the *ArrayList* object.
5. Count the number of *Auto* elements in the *ArrayList* with a specified model.

Table 9.6 provides the API for the *Auto* class.

Constructors and Some Methods of the *Auto* Class

***Auto* Class Constructor Summary**
`Auto( )` creates an *Auto* object with initial default values of "unknown," 0, and 0.0.
`Auto( String model, int milesDriven, double gallonsOfGas )` creates an *Auto* object with the initial values of *model, milesDriven,* and *gallonsOfGas.*

***Auto*Class Method Summary**

Return value	**Method name and argument list**
`String`	`getModel( )` returns the value of *model*
`int`	`getMilesDriven( )` returns the value of *milesDriven*
`double`	`getGallonsOfGas( )` returns the value of *gallonsOfGas*
`Auto`	`setModel( String model )` mutator for *model*
`Auto`	`setMilesDriven( int milesDriven )` mutator for *milesDriven*
`Auto`	`setGallonsOfGas(`

	`double gallonsOfGas )` mutator for *gallonsOfGas*
`double`	`milesPerGallon( )` returns the gas mileage for this *Auto* object
`String`	`toString( )` returns a *String* representation of this object
`boolean`	`equals( Object obj )` compares this *Auto* object to another object

Description

The framework for this Programming Activity will animate your algorithm so that you can check the accuracy of your code. For example, **Figure 9.17** shows the application counting the number of *Auto* elements in the *ArrayList* object having a *model* value equal to “Ferrari.” The application accesses each element in the *ArrayList* in order, checking the *model* for the desired value, “Ferrari.” At this point, the current element being accessed is a *BMW* and the application has found two *Auto* elements with the *model* value, “Ferrari.”

Figure 9.17 Animation of the Programming Activity

Description

Instructions

In this chapter’s Programming Activity 2 folder in the supplied code files, you will find the source files needed to complete this activity. Copy all the files to a folder on your computer. Note that all files should be in the same folder.

Open the *ArrayListController.java* source file. Searching for five asterisks (*****) will position you at the sample method and the four other locations where you will add your code. We have provided the sample code for task number 1. In every task, you will fill in the code for a method that will manipulate an existing *ArrayList* of *Auto* elements. You should not instantiate the *ArrayList* object; we have done that for you. Example 9.16

shows the section of the *ArrayListController* source code where you will add your code.

Note that for the *countFound* and *findMaximumMilesDriven* methods, we provide a dummy *return* statement (*return* 0;). We do this so that the source code will compile. In this way, you can write and test each method separately, using step-wise refinement. When you are ready to write the *countFound* and *findMaximumMilesDriven* methods, just replace the dummy *return* statements with the appropriate *return* statement for that method.

Example 9.16 Location of Student Code in *ArrayListController*

```java
// ***** 1.  This method has been coded as an example
/** Fills the carList with hard-coded Auto objects
*    The instance variable carList is the ArrayList
*        to be filled with Auto objects
*/
public void fillWithCars( )
{
  // clear carList before adding cars
  carList.clear( );
  // Reset the number of Autos to 0
  // This is needed so that the animation feedback works correctly
  Auto.clearNumberAutos( );

  Auto car1 = new Auto( "BMW", 0, 0.0 );
  Auto car2 = new Auto( "Ferrari", 100, 500.0 );
  Auto car3 = new Auto( "Jeep", 1000, 90.0 );
  Auto car4 = new Auto( "Ferrari", 10, 3.0 );
  Auto car5 = new Auto( "BMW", 4000, 200.0 );
  Auto car6 = new Auto( "Ferrari", 1000, 50.0 );

  carList.add( car1 );
  animate( car1 );
  carList.add( car2 );
  animate( car2 );
  carList.add( car3 );
  animate( car3 );
  carList.add( car4 );
  animate( car4 );
  carList.add( car5 );
  animate( car5 );
  carList.add( car6 );
  animate( car6 );
}
// end of fillWithCars method

// ***** 2.  Student writes this method
/**  Prints carList to console, elements are separated by a space
*    The instance variable carList is the ArrayList to be printed
*/
public void printAutoList( )
{
 // Note:  To animate the algorithm, put this method call as the
 // last statement in your for loop
 //              animate( car );
 //  where car is the variable name for the current Auto object
 //  as you loop through the ArrayList object
 // Write your code here:

}
// end of printAutoList method

// ***** 3.  Student writes this method
/** Sets the model of all the elements in carList to parameter value
* The instance variable carList is the ArrayList to be modified
*  @param model the model to assign to all Auto objects in carList
*/
public void setModelValues( String model )
{
 // Note:  To animate the algorithm, put this method call as the
 // last statement in your for loop
 //            animate( car );
 //  where car is the variable name for the current Auto object
 //  as you loop through the ArrayList object
 // Write your code here:

}
// end of setModelValues method

// ***** 4.  Student writes this method
/** Finds maximum number of miles driven
*   Instance variable carList is the ArrayList to search
*  @return     the maximum miles driven by all the Auto objects
*/
public int findMaximumMilesDriven( )
{
 // Note:  To animate the algorithm, put this method call as the
 // last statement in your for loop
 //           animate( car, maximum );
 //  where car is the variable name for the current Auto object
 //  and maximum is the int variable storing the current maximum
 //  number of miles for all Auto elements you have already tested
 //  as you loop through the ArrayList object
 // Write your code here:

 return 0; // replace this statement with your return statement

}
// end of findMaximumMilesDriven method

// ***** 5.  Student writes this method
/** Finds number of times parameter model is found in the carList
*   Instance variable carList is the ArrayList in which we search
*  @param model       the model to count
*  @return            the number of times model was found
*/
public int countFound( String model )
{
 // Note:  To animate the algorithm, put this method call as the
 // last statement in your for loop
 //             animate( car, num );
 //  where car is the variable name for the current Auto object
 //  and num is the int variable storing the current number of
 //  Auto elements whose model is equal to the method's parameter
 //  as you loop through the ArrayList object
 // Write your code here:

 return 0; // replace this statement with your return statement

}
// end of countFound method
```

Description

The framework will animate your code so that you can watch it work. For this to happen, be sure that your *for* loops call the *animate* method. The arguments that you send to *animate* are not always the same, but the location of the call to *animate* is always the same, that is, the last statement of your *for* loop. Detailed instructions for each task are included in the code.

To test your code, compile *ArrayListController.java* and run the *ArrayListApplication* source code. Figure 9.18 shows the graphics window when the program begins. Because the *Auto* elements of the *ArrayList* object are hard-coded, the values will be the same each time the program runs. Click on the “Fill Cars” button first. To test any method, click on the appropriate button.

Figure 9.18 The Graphics Window When the Application Begins

Description

Troubleshooting

If your method implementation does not animate, check these tips:

- Verify that the last statement in your *for* loop is a call to the *animate* method and that you passed the loop variable(s) as the argument(s), as in the following:

```
animate( car ); // or
animate( car, maximum ); // or
animate( car, num );
```

  Description

- Verify that the headers of your *for* loops are correct. They should all be the same.
- Verify that you updated the variables *maximum* and *num* correctly.

DISCUSSION QUESTIONS

1. Change the code in the *fillWithCars* method so that there are more or fewer *Auto* objects in the *ArrayList*. How does the number of *Auto* objects impact how the other methods are coded? Explain.
2. **Explain how looping through an** *ArrayList* **is different from looping through an array.**

```
states: {MD=Maine, NY=New York, NJ=New Jersey, CA=California}
NY is New York

states: {MD=Maryland, NY=New York, NJ=New Jersey, CA=California}
Maryland was removed

states: {NY=New York, NJ=New Jersey, CA=California}

NY: New York
NJ: New Jersey
CA: California

New York
New Jersey
California
```

9.9 The *HashMap* Class

Sometimes data comes in the form of (key, value) pairs. A data set can be made up of (key, value pairs) where each value is associated with a key and can be accessed via its key. For example, the keys could be U.S. states' abbreviated names (e.g., MD, CA, NY, ..) and their associated values could be the state's full name (Maryland, California, New York, ..). Or, the keys could be SUN, MON TUES, .. and the values could be Sunday, Monday, Tuesday, … Such data structures are called hash tables, or hash maps, or associative arrays. Other programming languages, like Python, call them dictionaries.

The *HashMap* class, part of the *java.util* package, implements such data structure. Like the *ArrayList* class, the *HashMap* class uses generics. Note that there is no concept of order in a *HashMap*. The only way to access a value is via its key; there is no index.

9.9.1 Declaring and Instantiating *HashMap* Objects

Here is the syntax for declaring a *HashMap* of objects:

```
HashMap<ClassName1, ClassName2> hashMapName;
```

Description

ClassName1 is the data type of the keys, and ClassName2 is the data type of the values. As with the *ArrayList* class, we cannot declare a *HashMap* using primitive data types. For example, these two statements declare two *HashMaps*:

```
HashMap<String, String> countries;
HashMap<Integer, Auto> cars;
```

Description

The first one stores a hash map whose keys are *Strings* and values are *String* objects.

The second one stores a hash map whose keys are *Integers* and values are *Auto* objects. Note that we cannot use a primitive data type, so we use *Integer*, the wrapper class for *int*.

Two constructors of the *HashMap* class are listed in **Table 9.7**.

HashMap Constructors

HashMap Constructors Summary

Constructor name and argument list

`HashMap<ClassName1, ClassName2>( )`

Constructs a *HashMap* object whose keys are *ClassName1* objects and values are *ClassName2* objects

`HashMap<ClassName1, ClassName2>( int initialCapacity )`

Constructs a *HashMap* object whose keys are *ClassName1* objects and values are *ClassName2* objects. The resulting *HashMap* object has the specified initial capacity and a default load factor of 0.75. A load factor of 0.75 means that if the capacity of the HashMap is 75% full, the capacity will be automatically increased.

Description

Here is the syntax for instantiating a *HashMap* of objects:

```
hashMapName = new HashMap <ClassName1, ClassName2>( argumentList );
```

Description

These two statements will instantiate the two *HashMap* objects declared earlier:

```
countries = new HashMap <String, String>( );
cars = new HashMap <String, Auto>( 10 );
```

Description

Countries is instantiated using the default constructor, and *cars* is instantiated with an initial capacity of 10. Instantiating a *HashMap* without specifying the object types will result in a compiler warning.

9.9.2 Methods of the *HashMap* Class

Among others, the *HashMap* class provides methods to do the following operations:

- add a (key, value) pair
- remove a (key, value) pair
- replace an existing value with a new value based on its key
- check if the *HashMap* is empty
- check if a key is present in the *HashMap*
- check if a value is present in the *HashMap*
- return the number of (key, value) pairs
- return all the values of the *HashMap* as a list

Some methods of the *HashMap* class are listed in **Table 9.8.**

Useful Methods of the *HashMap* Class

Useful Methods of the *HashMap* Class

Return value	**Method name and argument list**
`V`	`put( K key, V value )` Adds the (*key*, *value*) pair

	specified to this *HashMap*. If this *HashMap* previously contained *key*, the old value is replaced and returned. Returns *null* if there was no previous mapping for *key*. *K* and *V* are the data types for the keys and the values of this *HashMap*, respectively.
`V`	`get( K key )` Returns the value mapped to *key*, *null* if there is no value mapped to *key*.
`boolean`	`isEmpty( )` Returns *true* if this *HashMap* contains no mapping, *false* otherwise.
`V`	`remove( K key )` Removes the (*key*, value) pair if *key* is found and returns the value mapped to *key*. Returns *null* if no mapping exists for *key*.
`boolean`	`replace( K key, V oldValue, V newValue )` Changes *oldValue* to *newValue* if *oldValue* is mapped to *key* in this *HashMap*; if it is, it returns *true*, *false* otherwise.
`int`	`size( )` Returns the number of (key, value) pairs in this *HashMap*.
`boolean`	`contains( K key )` Returns *true* if *key* is a key in this *HashMap*, *false* otherwise.
`boolean`	`contains( V value )` Returns *true* if *value* is mapped to one or more keys in this *HashMap*, *false* otherwise.
`Set<K>`	`keySet( )` Returns a *Set* of keys contained in this *HashMap*.
`Collection<V>`	`values( )`

	Returns a *Collection* of values contained in this *HashMap*.

Description

9.9.3 Using the *HashMap* in a Program

Example 9.17 shows how we can use the *HashMap* class to store states.

Example 9.17 Using *HashMap* Methods

```
 1 /* A simple HashMap of ( String, String ) pairs
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.HashMap;
 6 import java.util.Set;
 7
 8 public class HashMapStates
 9 {
10   public static void main( String [ ] args )
11   {
12     HashMap<String,String> states = new HashMap<String,String>( );
13     states.put( "CA", "California" );
14     states.put( "MD", "Maine" ); // We correct this later
15     states.put( "NY", "New York" );
16     states.put( "NJ", "New Jersey" );
17
18     // Output all (key, value) pairs
19     System.out.println( "states: " + states );
20
21     // Retrieve value whose key is NY
22     String nyState = states.get( "NY" );
23     System.out.println( "NY is " + nyState );
24
25     // Change value whose key is MD to Maryland
26     states.replace( "MD", "Maryland" );
27     System.out.println( "states: " + states + "\n" );
28
29     // Remove state whose key is MD
30     String mdState = states.remove( "MD" );
31     System.out.println( mdState + " was removed" );
32     System.out.println( "states: " + states + "\n" );
33
34     // Loop through all the values using the keys
35     Set<String> keys = states.keySet( );
36     for( String abbr: keys )
37       System.out.println( abbr + ": " + states.get( abbr ) );
38     System.out.println( );
39
40     // Loop through all the values directly
41     for( String state: states.values( ) )
42       System.out.println( state );
43   }
44 }
```

Description

At line 12, we declare and instantiate the *HashMap states*. At lines 13–16, we place four (key, value) pairs of states in *states*. We output *states* at lines 18–19 by implicitly calling *toString*. Note that the *toString* method of *HashMap* returns a nice representation of the *HashMap*. We retrieve and output the value associated with the key *NY* at lines 21–23. At lines 25–26, we replace *Maine*, the old value for the key *MD*, with *Maryland*.

We then output *states* at line 27 to verify that the change has taken place. At lines 29–30, we remove the (key, value) pair associated with the key *MD* and capture the returned value. We output the returned value (*Maryland*) and the states again at lines 31–32, showing that (*MD*, *Maryland*) is no longer in *states*.

At lines 34–37, we loop through all the keys in states and output each key and its corresponding value. We first call the *keySet* method at line 35 to retrieve the keys as a *Set* of *Strings*. At lines 36–37, we loop through them using the enhanced *for* loop and output all the (key, value) pairs.

At lines 40–42, we call the *values* method and loop through the *Collection* of values returned by it in order to output them all. **Figure 9.19** shows the output in Example 9.17.

Figure 9.19 Output of Example 9.17

Description

with these end-of-chapter questions

9.11.1	Multiple Choice Exercises Questions 9, 10, 11, 12, 13
9.11.2	Reading and Understanding Code Questions 29, 30, 31, 32
9.11.3	Fill In the Code Questions 46, 47, 48, 49
9.11.4	Identifying Errors in Code Questions 55, 56, 57, 58
9.11.5	Debugging Area Questions 63, 64
9.11.6	Write a Short Program Questions 80, 81, 82
9.11.8	Technical Writing Question 98

© Lois GoBe/Shutterstock.

CHAPTER REVIEW

9.10 Chapter Summary

- Arrays can be single-dimensional, two-dimensional, three-dimensional, or more generally, *n*-dimensional.
- In a two-dimensional array, each row is an array.
- Each element in a two-dimensional array is accessed using the array name with a row index and column index that refer to the element’s position in the array.
- Concepts such as declaration, instantiation, initial values, indexing, and aggregate operations from single-dimensional arrays also apply to two-dimensional arrays.
- Two-dimensional arrays can be instantiated by assigning initial values in a comma-separated list of comma-separated lists at the declaration.
- Each row in a two-dimensional array can have a different number of columns.
- A two-dimensional array has an instance variable, *length*, which holds the number of rows in the array.
- Each row of a two-dimensional array has an instance variable, *length*, which holds the number of elements in that row.
- The *ArrayList* and *HashMap* classes implement generics and are part of the *java.util* package.
- An *ArrayList* can be thought of as an expandable single-

dimensional array of objects.

- To define an *ArrayList* to hold elements of primitive data types, use the wrapper classes.
- An *ArrayList* object expands automatically as objects are added.
- We access an element of an *ArrayList* via its index.
- We can process each element in an *ArrayList* using the enhanced *for* loop.
- A *HashMap* can be thought of as a set of (key, value) pairs.
- We can access a value in a *HashMap* via its key.
- We can retrieve all the keys or all the values in a *HashMap* using the *keySet* and *values* methods. We can then loop through them using an enhanced *for* loop.

9.11 Exercises, Problems, and Projects

9.11.1 Multiple Choice Exercises

1. What is/are the valid way(s) to declare a two-dimensional integer array named *a*? (Check all that apply.)

   - `int [ ][ ] a;`
   - `int a[ ][ ];`
   - `array[ ]` int `a;`
   - `int array[] a;`

6. A two-dimensional array is an array of arrays.

   - true
   - false

9. In a two-dimensional array, every row must have the same number of columns.

   - true
   - false

12. What is the default value of the elements of a two-dimensional array of *booleans* after declaration and instantiation of the array?

   - *true*
   - *false*
   - undefined

16. How do you access the element of array *a* located at row 2 and column 4?

   - `a{2}{4}`
   - `a( 2,4 )`

- `a[2][4]`
- `a[4][2]`

21. How do you retrieve the number of rows in a two-dimensional array *a*?

    - `a.rows`
    - `a.length`
    - `a.rows()`
    - `a.size`

26. How do you retrieve the number of columns in row 2 in a two-dimensional array *a*?

    - `a.length`
    - `a[2].length`
    - `a.size`
    - `a[2].size`

31. All the elements of a two-dimensional array must be of the same type.

    - true
    - false

34. An *ArrayList* can be returned by a method.

    - true
    - false

37. It is possible to declare and instantiate an *ArrayList* of a user-defined class type.

    - true
    - false

40. As we add objects to an *ArrayList*, how can we be sure it has enough capacity?

- Use the *setCapacity* method.
- Use the *trimToSize* method.
- We don’t need to do anything; capacity expands automatically as needed.

44. Where does the *add* method of the *ArrayList* class add an object?

    - at the beginning of the list
    - at the end of the list

47. To what package does the class *HashMap* belong?

    - *java.io*
    - *java.util*
    - *java.array*
    - *java.list*

9.11.2 Reading and Understanding Code

For Questions 14 to 24, consider the following two-dimensional array declaration and initialization:

```
String [ ][ ] cities = { { "New York", "LA", "San Francisco", "Chicago" },
                         { "Munich", "Stuttgart", "Berlin", "Bonn" },
                         { "Paris", "Ajaccio", "Lyon" },
                         { "Montreal", "Ottawa", "Vancouver" } };
```

Description

1. How many rows are in the array *cities*?
2. What is the value of the expression *cities[2][1]*?
3. What is the index of the last row in the array *cities*?
4. What are the row and column indexes of *Chicago* in the array *cities*?
5. What is the output of this code sequence?

```
System.out.println( cities[3][2] );
```

Description

6. What is the output of this code sequence?

```
for ( int j = 0; j < cities[1].length; j++ )
   System.out.println( cities[1][j] );
```

Description

7. What is the output of this code sequence?

```
for ( int i = 0; i < cities.length; i++ )
   System.out.println( cities[i][1] );
```

Description

8. What is the output of this code sequence?

```
for ( int i = 0; i < cities.length; i++ )
{
    for ( int j = 0; j < cities[i].length; j++ )
        System.out.print( cities[i][j] + "\t" );
    System.out.println( );
}
```

Description

9. What is the output of this code sequence?

```
for ( int i = 0; i < cities.length; i++ )
{
   for ( int j = 0; j < cities[i].length; j++ )
   {
         if ( cities[i][j].length( ) == 6 )
             System.out.println( cities[i][j] );
   }
}
```

Description

10. What is the output of this code sequence?

```
int count = 0;
for ( int i = 0; i < cities.length; i++ )
{
   for ( int j = 0; j < cities[i].length; j++ )
   {
         if ( cities[i][j].length( ) == 7 )
               count++;
   }
}
System.out.println( "count is " + count );
```

Description

11. What is the output of this code sequence?

```
for ( int i = 0; i < cities.length; i++ )
{
   for ( int j = 0; j < cities[i].length; j++ )
   {
        if ( cities[i][j].charAt( 0 ) == 'S' )
              System.out.println( cities[i][j] );
   }
}
```

Description

12. What does this method do?

```
public static int foo( double [ ][ ] a )
{
   int b = 0;
   for ( int i = 0; i < a.length; i++ )
   {
      for ( int j = 0; j < a[i].length; j++ )
            b++;
   }
   return b;
}
```

Description

13. What does this method do?

```
public static boolean foo( char [ ][ ] a )
{
   int b = a[0].length;
   for ( int i = 1; i < a.length; i++ )
   {
         if ( a[i].length != b )
            return false;
   }
   return true;
}
```

Description

14. What does this method do?

```
public static int foo( String [ ][ ] a )
{
      int b = 0;
      for ( int i = 0; i < a.length; i++ )
            b++;
      return b;
}
```

Description

15. What does this method do?

```
public static int [ ] foo( float [ ][ ] a )
{
      int [ ] temp = new int [a.length];
      for ( int i = 0; i < a.length; i++ )
            temp[i] = a[i].length;
      return temp;
}
```

Description

16. What does this method do?

```
public static int foo( ArrayList<Integer> a )
{
      int b = 0;
      for ( Integer i : a )
          b++;
      return b;
}
```

Description

17. After the following code sequence is executed, what are the contents and index of each element of *a*?

```
ArrayList<Integer> a = new ArrayList<Integer>( );
a.add( 7 );
a.add( 4 );
a.add( 21 );
```

Description

18. After the following code sequence is executed, what are the contents and index of each element of *a*?

```
ArrayList<Integer> a = new ArrayList<Integer>( );
a.add( 7 );
a.add( 4 );
a.add( 21 );
a.set( 1, 45 );
```

Description

19. After the following code is executed, what are the (key, value) pairs in *days*?

```
HashMap<String, String> days = new HashMap<String, String>( );

days.put( "SUN", "Sunday" );
days.put( "MON", "Monday" );
days.put( "TUES", "Tuesday" );
```

Description

9.11.3 Fill In the Code

For Questions 33 to 37, consider the following statement:

```
String [ ][ ] geo = { { "MD", "NY", "NJ", "MA", "ME", "CA", "MI", "OR" },
                      { "Detroit", "Newark", "Boston", "Seattle" } };
```

Description

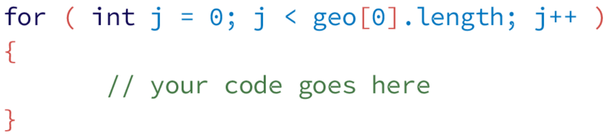
1. This code prints the element at row index 1 and column index 2 of the two-dimensional array *geo*.

```
// your code goes here
```

2. This code prints the element of the array *geo* whose value is “CA.”

```
// your code goes here
```

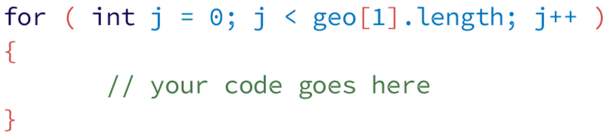
3. This code prints all the states (i.e., the first row) that start with an *M* in the array *geo*.



```
for ( int j = 0; j < geo[0].length; j++ )
{
        // your code goes here
}
```

Description

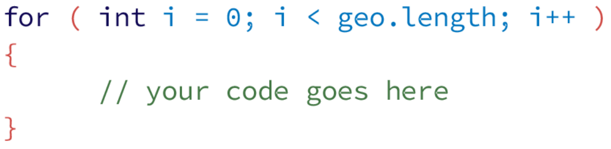
4. This code prints all the cities (i.e., the second row) in the array *geo*.



```
for ( int j = 0; j < geo[1].length; j++ )
{
        // your code goes here
}
```

Description

5. This code prints all the elements of the array *geo*.



```
for ( int i = 0; i < geo.length; i++ )
{
        // your code goes here
}
```

Description

For Questions 38 to 41, consider the following statement:

```
int [ ][ ] a = { { 9, 6, 8, 10, 5 },
                 { 7, 6, 8, 9, 6 },
                 { 4, 8, 10, 6, 6 } };
```

Description

1. This code calculates and prints the sum of all the elements in the array *a*.

   ```
   int sum = 0;
   for ( int i = 0; i < a.length; i++ )
   {
         // your code goes here
   }
   System.out.println( "sum is " + sum );
   ```

   Description

2. This code counts and prints the number of times the value 8 appears in the array *a*.

   ```
   int count = 0;
   for ( int i = 0; i < a.length; i++ )
   {
         // your code goes here
   }
   System.out.println( "# of 8s in a: " + count );
   ```

   Description

3. This code counts and prints the number of times the value 6 appears in the second row (i.e., the row whose index is 1) of array *a*.

   ```
   int count = 0;

   // your code for the for loop header goes here
   {
          if ( a[1][j] == 6 )
               count++;
   }
   System.out.println( "# of 6s in the 2nd row: " + count );
   ```

Description

4. This code calculates the sum of the elements in the second column (i.e, the column with index 1) of array *a*.

```
int sum  = 0;
for ( int i = 0; i < a.length; i++ )
{
   // your code goes here

}
System.out.println( "sum is " + sum );
```

Description

5. This method returns *true* if an element in an array of *Strings* is equal to “Java”; otherwise, it returns *false*.

```
public static boolean foo( String [ ][ ] a )
{
   // your code goes here
}
```

Description

6. This method returns the product of all the elements in an array.

```
public static int foo( int [ ][ ] a )
{
   // your code goes here
}
```

Description

7. This method returns *true* if there is at least one row in the array that has exactly five columns; otherwise, it returns *false*.

```
public static boolean foo( char [ ][ ] a )
{
   // your code goes here
}
```

Description

8. This method takes a *HashMap* parameter of *Integer* keys and *String* values and returns an *ArrayList* made up of all the values in that *HashMap* parameter.

```
public static ArrayList<String> foo( HashMap<String, String> )
{
   // your code goes here

}
```

Description

For Questions 46 to 49, consider the following statements:

```
ArrayList<String> languages = new ArrayList<String>( );
languages.add( "SQL" );
languages.add( "Java" );
languages.add( "HTML" );
languages.add( "PHP" );
languages.add( "Perl" );
```

Description

1. This code prints the number of elements in *languages*.

```
// your code goes here
```

2. This code retrieves the *String* “HTML” from *languages* (without deleting it) and assigns it to the *String* variable *webLanguage*.

```
// your code goes here
```

3. This code replaces “HTML” *with* “C++” in *languages*.

```
// your code goes here
```

4. This code prints all the elements of *languages* that start with the letter *P*.

   ```
   for ( String s : languages )
   {
     // your code goes here
   }
   ```

   Description

9.11.4 Identifying Errors in Code

1. Where is the error in this code sequence?

   ```
   double [ ][ ] a = { 3.3, 26.0, 48.4 };
   ```

   Description

2. Where is the error in this code sequence?

   ```
   int [ ][ ] a = { { 3, 26, 4 }, { 14, 87 } };
   System.out.println( a[1][2] );
   ```

   Description

3. Where is the error in this code sequence?

   ```
   double [ ][ ] a = new double [ ][10];
   ```

   Description

4. Where is the error in this code sequence?

   ```
   int [ ][ ] a = { { 1, 2 },
                    { 10.1, 10.2 } };
   ```

   Description

5. Where is the error in this code sequence? (This code compiles and runs, but does not output the array values.)

```
int [ ][ ] a = { { 3, 26, 48 }, { 5, 2, 9 } };
System.out.println( "The array elements are " + a );
```

Description

6. Where is the error in this code sequence?

```
ArrayList<double> al;
```

Description

7. Where is the error in this code sequence?

```
ArrayList<Float> a1 = new ArrayList( )<Float>;
```

Description

8. Where is the error in this code sequence?

```
ArrayList<Double> a;
a = new ArrayList<Float>( );
```

Description

9. Where is the error in this code sequence?

```
// a is an ArrayList of Strings
// a has already been declared and instantiated
a.size( ) = 10;
```

Description

9.11.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. You coded the following on line 14 of the *Test.java* class:

```
int a[2][ ] = { { 2, 7 }, { 9, 2 } }; // line 14
```

Description

When you compile, you get the following message:

```
Test.java:14: error: ']' expected
     int a[2][ ] = { { 2, 7 }, { 9, 2 } }; // line 14
           ^
Test.java:14: error: not a statement
     int a[2][ ] = { { 2, 7 }, { 9, 2 } }; // line 14
                       ^
Test.java:14: error: ';' expected
     int a[2][ ] = { { 2, 7 }, { 9, 2 } }; // line 14
                        ^
Test.java:14: error: illegal start of expression
     int a[2][ ] = { { 2, 7 }, { 9, 2 } }; // line 14
                             ^
Test.java:14: error: not a statement
     int a[2][ ] = { { 2, 7 }, { 9, 2 } }; // line 14
                                 ^
Test.java:14: error: ';' expected
     int a[2][ ] = { { 2, 7 }, { 9, 2 } }; // line 14
                                  ^
6 errors
```

Description

Explain what the problem is and how to fix it.

2. You coded the following in the *Test.java* class:

```
int [ ][ ] a = { { 1, 2, 3, 4 },
                 { 10, 20, 30 } };
for ( int i = 0; i < a.length; i++ )
{
      for ( int j = 0; j < a[0].length; j++ )
      {
            System.out.println( a[i][j] ); // line 14
      }
}
```

Description

The code compiles properly but when you run, you get the following output:

```
 1
 2
 3
 4
10
20
30
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
       Index 3 out of bounds for length 3
       at Test.main(Test.java: 14)
```

Description

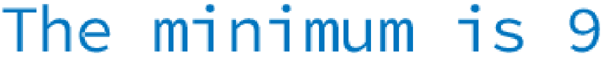
Explain what the problem is and how to fix it.

3. You coded the following in the *Test.java* class in order to output the smallest element in the array *a*:

```
int [ ][ ] a = { { 9, 8, 7, 6 },
                 { 10, 20, 30, 40 } };
int min = a[0][0];
for ( int i = 1; i < a.length; i++ )
{
      for ( int j = 0; j < a[i].length; j++ )
      {
            if ( a[i][j] < min )
                 min = a[i][j];
      }
}
System.out.println( "The minimum is " + min );
```

Description

The code compiles properly, but when you run, you get the following output:



```
The minimum is 9
```

Description

You expected the value of *min* to be 6. Explain what the problem is and how to fix it.

4. You coded the following in file *Test.java*:

```
int [ ][ ] a = { { 9, 8, 7, 6 },
                 { 10, 20, 30, 40 } };
for ( int j = 0; j <= a[1].length; j++ )
{
  if ( a[1][j] == 20 ) // line 14
  {
     System.out.println( "Found 20 at column index " + j
                         + " of second row" );
  }
}
```

Description

The code compiles properly, but when you run, you get the following output:

```
Found 20 at column index 1 of second row
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
        Index 4 out of bounds for length 4
        at Test.main(Test.java:14)
```

Description

Explain what the problem is and how to fix it.

5. You coded the following in the *Test.java* class:

```
// cars is an ArrayList of Auto objects
// cars has already been declared and instantiated
for ( Auto a ; cars )     // line 12
{
    System.out.println( a.toString( ) );
}   // line 15
```

Description

When you compile, you get the following message :

```
Test.java:12: error: ';' expected
  for ( Auto a ; cars ) // line 12
                     ^

1 error
```

Description

Explain what the problems are and how to fix them.

6. You coded the following in the *Test.java* class:

```
ArrayList<String> a = new ArrayList<String>( );
a.add( "Cloudy" );
a.add( "Snowy" );
a.add( "Cloudy" );
System.out.println( "Weather is " + a.get( 3 ) ); // line 14
```

Description

The code compiles properly, but when you run, you get the following output:

```
Exception in thread "main" java.lang.IndexOutOfBoundsException:
Index 3 out of bounds for length 3
...
```

Description

Explain what the problem is and how to fix it.

7. You coded the following in the file *Test.java*:

```
ArrayList<Integer> a = new ArrayList( );
```

Description

When you compile (using Xlint), you get the following warning message:

```
Test.java:10: warning: [rawtypes] found raw type: ArrayList
    ArrayList<Integer> a = new ArrayList( );
                               ^
    missing type arguments for generic class ArrayList<E>
    where E is a type-variable:
    E extends Object declared in class ArrayList

Test.java:10: warning: [unchecked] unchecked conversion
    ArrayList<Integer> a = new ArrayList( );
                           ^
  required: ArrayList<Integer>
  found:    ArrayList
2 warnings
```

Description

Explain what the problem is and how to fix it.

8. You coded the following in the file *Test.java*:

```
a.add( 2.3 );
a.add( 8.4 );
a.add( 5 ); // line 11
```

Description

When you compile, you get the following message:

```
Test.java:11: error: incompatible types: int cannot be converted to Double
    a.add( 5 ); // line 11
           ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose
    to get full output
1 error
```

Description

Explain what the problem is and how to fix it.

9. You coded the following in the file *Test.java*:

```
ArrayList<Character> a = new ArrayList<Character>( );
a.add( 'X' );
a.add( 'A' );
a.add( 'V' );
a.add( 'A' );
a.set( 1, 'J' );
for ( Character c : a )
  System.out.print( c + " " );
```

Description

The code compiles properly, but when you run, you get the following output:

```
X J V A
```

when you expected:

```
J A V A
```

Explain what the problem is and how to fix it.

9.11.6 Write a Short Program

1. Write a value-returning method that returns the number of rows in a two-dimensional array of *doubles*. Include code to test your method.
2. Write a value-returning method that returns the number of elements in a two-dimensional array of *floats*. Include code to test your method.
3. Write a value-returning method that returns the number of columns that have two elements in a two-dimensional array of *booleans*. Include code to test your method.
4. Write a value-returning method that returns the number of columns with *n* elements in a two-dimensional array of *chars*, where *n* is a parameter of the method. Include code to test your method.
5. Write a value-returning method that returns the sum of all the elements in a two-dimensional array of *floats*. Include code to test your method.
6. Write a method with a *void* return value that sets to 0 all the elements of the even-numbered rows and sets to 1 all the elements of odd-numbered rows of a two-dimensional array

of *ints*. Include code to test your method.

7. Write a value-returning method that returns the sum of the elements in the last column of each row in a two-dimensional array of *ints*. Include code to test your method.
8. Write a method with a *void* return value that inverts all the elements of a two-dimensional array of *booleans* (*true* becomes *false* and *false* becomes *true*). Include code to test your method.
9. Write a method that returns the number of elements having the value *true* in a two-dimensional array of *booleans*. Include code to test your method.
10. Write a method that returns the percentage of elements having the value *false* in a two-dimensional array of *booleans*. Include code to test your method.
11. Write a method that returns the average of all elements in a two-dimensional array of *ints*. Include code to test your method.
12. Write a method that returns the *String* “regular” if all the rows of a two-dimensional array of *floats* have the same number of columns; otherwise, it returns “irregular.” Include code to test your method.
13. Write a method that returns the concatenation of all elements in a two-dimensional array of *Strings*. Include code to test your method.
14. Write an array-returning method that takes a two-dimensional array of *chars* as a parameter and returns a single-dimensional array of *Strings* as follows: The array returned should have a number of elements equal to the number of rows in the parameter array; every element of the array returned should be the concatenation of all the column elements of the corresponding row in the

parameter array. Include code to test your method.

15. Write a method that takes as parameters two *ArrayLists* of *Strings* of the same size. The method returns a *Hashtable* (look up the *Hashtable* class in the Java Class Library) such that the *Hashtable*'s keys are the elements of the first *ArrayList*, and the *Hashtable*'s values are the elements of the second *ArrayList*. Test your method in a program using an *ArrayList* of state codes and an *ArrayList* of state names.
16. Write a method that returns the sum of all the elements of an *ArrayList* of *Integer* objects. Include code to test your method.
17. Write a method that returns the *String* “odd” or “even” if the number of elements of an *ArrayList* of *Strings* is odd or even. Include code to test your method.
18. Write a method that takes an *ArrayList* of *Integer* objects and returns an *ArrayList* of *Character* objects of the same size. The returned elements of the *ArrayList* are assigned a letter grade corresponding to the integer grade of the same index element of the *ArrayList* parameter (A if 90 or above, ..., F if less than 60). Include code to test your method.

9.11.7 Programming Projects

1. Write a class (and a client class to test it) that encapsulates statistics for summer job salaries for a group of people over several years. Your only instance variable should be a two-dimensional array of values representing salaries. Dimension 1 represents the people and dimension 2 represents the year of the summer job. Your constructor can simply take two integers representing the number of people and the number of years, then randomly generate

the salaries and fill the array. You should include the following methods:

- a method returning the index of the person having made the most money over the years
- a method returning the year when the highest salary was earned
- a method returning the total amount of money made by all the people over the years

5. Write a class (and a client class to test it) that encapsulates the evolution of the passwords of three students over four months. Your only instance variable should be a two-dimensional array of values representing the passwords. Dimension 1 represents the student and dimension 2 represents the month. (Since we are concerned about security, we are assuming that people change their password once a month; we only care about the value of the password at the end of a given month.) Your constructor can simply take a single-dimensional array of words representing the 12 passwords; they can be assigned to the two-dimensional array elements one at a time, starting with the first row. You should include the following methods:

   - a method returning the index of the person who changed his or her password the most times
   - a method returning the longest password
   - a method changing all the passwords to “unlock”
   - a method returning *true* if at least one person had a given word—the method’s parameter—as his/her password in at least one month; *false* otherwise

10. Write a class (and a client class to test it) that encapsulates the evolution of the sales tax rates in the 50 U.S. states over the last 10 years. Your only instance variable should be a two-dimensional array of values representing the sales tax rates. Dimension 1 represents the state and dimension 2 represents the year. Your constructor can simply be a default constructor, randomly generating the sales tax rates, which should be between 0 and 0.06. You should include the following methods:

    - a method returning the index of the state that has the biggest average tax rate over the years
    - a method returning an array of indexes of the states that have had at least one year with a tax rate less than 0.001
    - a method returning the highest sales tax rate over the years for a given state (which will be a parameter)

14. Write a class (and a client class to test it) that encapsulates the evolution of the quality ratings of various hotels over the years. Hotel ratings are represented by a number of stars, which can vary from one star (lowest quality) to five stars (highest quality). Your only instance variable should be a two-dimensional array of values representing the quality ratings. Dimension 1 represents the hotel and dimension 2 represents the year. Your constructor can take two parameters representing the number of hotels and the number of years. The ratings can simply be generated randomly. You should include the following methods:

    - a method returning an array of indexes of the hotels that have earned five stars at least once over the years

- a method returning the average rating of all the hotels over the years
- a method printing the indexes of the hotels that have earned five stars every year
- a method returning *true* if at least one hotel earned five stars for at least one year; *false* otherwise

19. Write a class (and a client class to test it) that encapsulates the value of the 26 letters of the English alphabet in the game of Scrabble in 10 countries. You should have three instance variables:

    - a two-dimensional array of integers representing the point values of the letters in the various countries
    - a single-dimensional array representing the alphabet from a to z
    - another single-dimensional array representing 10 countries

    For the two-dimensional array, dimension 1 represents the letter and dimension 2 represents the country. Your constructor can simply be a default constructor, randomly generating the values between 1 and 10. You should include the following methods:

    - a method returning an array of letters with their highest point value in any country
    - a method printing the names of the countries that have at least one letter with a point value of 10
    - a method taking a *String* as a parameter and printing the score of the word represented by that *String* in every country

26. Write a class (and a client class to test it) that encapsulates

the numbers of the various chessboard pieces in a chess game. You should have two instance variables:

- a two-dimensional array of integers; each array element represents how many of a particular chess piece of a particular color are on the board. In order to set it up, consider the following:
  - The first dimension represents the color of the pieces. On a chessboard, there are white and black pieces.
  - The second dimension represents the pieces themselves. On a chessboard, we have on each side: one king, one queen, two bishops, two knights, two rooks, and eight pawns.
- a single-dimensional array describing the pieces (king, queen, etc.)

Your constructor can simply be a default constructor, declaring and instantiating the two arrays to match the preceding information. You should include the following methods:

- a method with a *void* return value, called *playerATakesPlayerB*, updating the array based on a piece being taken by the opponent. It takes two parameters:
  - a *boolean* parameter representing whether “white takes black” or “black takes white”
  - an *int* parameter representing which piece gets taken
- a method returning how many of a particular piece are on the board (this method takes a parameter representing the piece)

- a method taking a *boolean* as a parameter, representing a color and returning the value of the board for that particular color. You can consider that a king is worth 0 points, a queen is worth 6 points, a rook is worth 4 points, a knight and a bishop are each worth 3 points, and a pawn is worth 1 point

36. Write a class (and a client class to test it) that encapsulates a deck of cards. A deck of cards is made up of 52 cards. You should have three instance variables:

    - a two-dimensional array of values representing the cards
    - a single-dimensional array describing the suit: spades, hearts, diamonds, and clubs
    - an instance variable representing the trump suit

    For the two-dimensional array, dimension 1 represents the suit and dimension 2 represents the type of card (ace, two, three, ..., jack, queen, king). Your constructor should take one parameter, which will represent the suit of the trump. Based on that, the cards should be given the following values:

    - Non-trump from 2 to 10: 1 point
    - Non-trump jack = 2
    - Non-trump queen = 3
    - Non-trump king = 4
    - Non-trump ace = 5
    - Any trump card = Non-trump value + 1

    You should include the following methods:

    - a method returning the trump suit, by name

- a method printing the whole deck of cards, suit by suit, with the value for each card
- a method taking a *String* as a parameter representing a suit, and returning the total value of the cards of that suit

49. Write a class (and a client class to test it) that encapsulates a tic-tac-toe board. A tic-tac-toe board looks like a table of three rows and three columns partially or completely filled with the characters X and O. At any point, a cell of that table could be empty or could contain an X or an O. You should have one instance variable, a two-dimensional array of values representing the tic-tac-toe board.
    Your default constructor should instantiate the array so that it represents an empty board.
    You should include the following methods:

    - a method, returning a *boolean*, simulating a play with three parameters as follows: If the first parameter is *true*, then X is playing; otherwise, O is playing. The other two parameters represent what cell on the board is being played. If the play is legal, that is, the cell is a legal cell on the board and is empty, then the method should update the array and return *true*; otherwise, the array should not be updated and the method should return *false*
    - a method returning how many valid plays have been made so far
    - a method checking if a player has won based on the contents of the board; this method takes no parameter. It returns X if the “X player” has won, O if the “O player” has won, T if the game was a tie. A player wins if he or she has placed an X (or an O) in all cells in a

row, all cells in a column, or all cells in one of the two diagonals

53. Code a class and a client class to test it that includes the following:

    - a *HashMap* instance variable of *Character* keys and *String* values.
    - a default constructor that initializes all the *HashMap* keys to the hexadecimal characters in the alphabet (from 0 to F) and the values to their corresponding binary representations. For example, 'C' should be the key for the value "1100".
    - a method that accepts a *String* representing a hexadecimal number and returns its binary representation as a *String*; for example, if the argument passed to that method is "C3", the returned value should be "11000011". You should use the *HashMap* instance variable to perform the conversion.
    - a method that returns a *HashMap* that is the inverse of the *HashMap* instance variable; that is, it maps a 4-binary-digit *String* key to a *Character* value. For example, "1110" will be the key for the value 'E'. Do not hard code the *HashMap* returned by this method. Instead, derive it from the *HashMap* instance variable without knowing the (key, value) pairs that are in it.
    - a method that accepts two parameters: a *String* representing a binary number and a *HashMap* that can be used to convert the *String* parameter representing the binary number to a hexadecimal number; it returns a *String* representing the hexadecimal equivalent of the parameter. For example, if the arguments passed to that method are "11000011" and an appropriate *HashMap* (like the one returned by the method above), the returned value should be "C3".

    Inside the main method, test all the methods above.

Description

59. Write a *Garage* class (and a client class to test it) with one instance variable: an *ArrayList* of *Autos* (you can use the *Auto* class from this chapter).

    You should include the following methods:

    - a method returning the average number of miles of all cars in the garage
    - a method returning “full” if the garage has 100 cars or more, “below minimum” if the garage has fewer than 25 cars, and “normal load” if the garage has between 25 and 100 cars in it
    - a method returning the total number of gallons of gas used by all cars in the garage

63. Write a *ComputerPart* class and a *ComputerKit* class (and a client class to test them).

The *ComputerPart* class has two instance variables: a *String* representing an item (for instance, “cpu” or “disk drive”), and a *double* representing the price of that item. The *ComputerKit* class has just one instance variable: an *ArrayList* of *ComputerPart* objects (they make up a computer) representing the list of parts for the computer kit. You should include the following methods:

- a method returning “expensive” if the total of the prices of the *ComputerPart* objects is greater than 1,000, “cheap” if it is less than 250, “normal” if it is between 250 and 1,000
- a method returning *true* if a certain item is included in the list of parts; *false* otherwise
- a method returning how many times a particular item (for instance, “cpu,” or “memory”) is found in the list of parts

9.11.8 Technical Writing

1. A two-dimensional array can have a different number of columns in every row. Do you see that as an advantage or a disadvantage? Discuss.
2. Discuss the pros and cons of using an array vs. using an *ArrayList*.

9.11.9 Group Project (for a group of 1, 2, or 3 students)

1. Design and code a program including the following classes, as well as a client class to test all the methods coded:

   A *Passenger* class, encapsulating a passenger. A passenger has two attributes: a name, and a class of service, which will be 1 or 2.

   A *Train* class, encapsulating a train of passengers. A train of passengers has one attribute: a list of passengers, which must be represented with an *ArrayList*. Your constructor will build the list of passengers by reading data from a file called *passengers.txt*. You can assume that *passengers.txt* has the following format:

<name1> <class1>
<name2> <class2>
...

For instance, the file could contain:

James	1
Ben	2
Suri	1
Sarah	1
Jane	2

...

You should include the following methods in your *Train* class:

- a method returning the percentage of passengers traveling in first class
- a method taking two parameters representing the price of traveling in first and second class and returning the total revenue for the train
- a method checking if a certain person is on the train; if he/she is, the method returns *true*; otherwise, it returns *false*

© Lois GoBe/Shutterstock.

CHAPTER 10

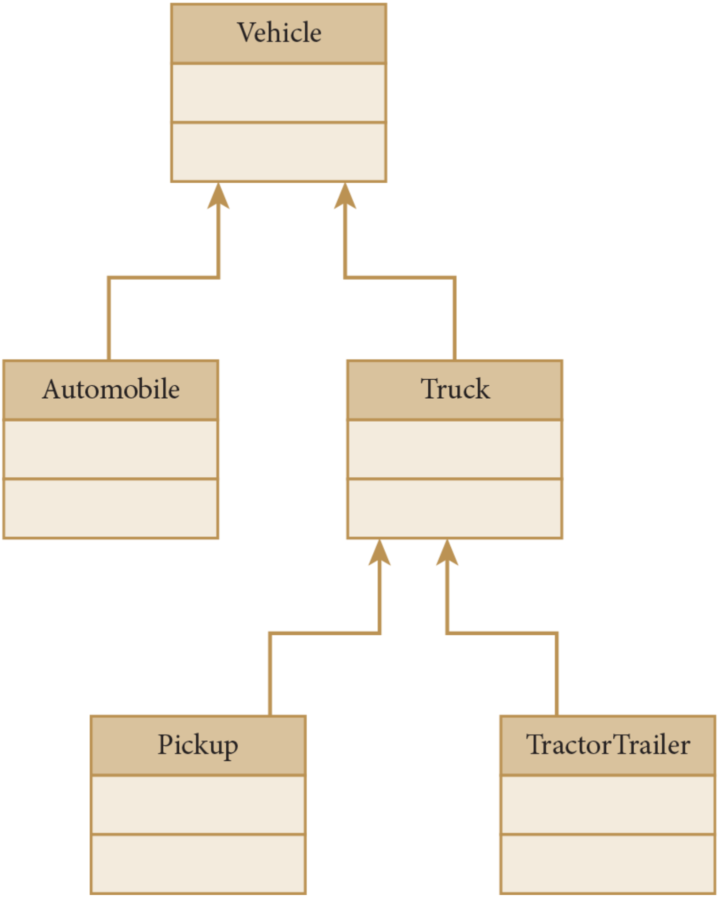
Object-Oriented Programming, Part 3: Inheritance, Polymorphism, and Interfaces

CHAPTER CONTENTS

- Introduction
- 10.1 Inheritance
- 10.2 Inheritance Design
    - 10.2.1 Inherited Members of a Class
    - 10.2.2 Subclass Constructors
    - 10.2.3 Adding Specialization to the Subclass
    - 10.2.4 Overriding Inherited Methods
- 10.3 The *protected* Access Modifier
- 10.4 **Programming Activity 1: Using Inheritance**
- 10.5 *Abstract* Classes and Methods
- 10.6 Polymorphism

10.7 **Programming Activity 2: Using Polymorphism**
10.8 Interfaces
10.9 Chapter Summary
10.10 Exercises, Problems, and Projects

  - 10.10.1 Multiple Choice Exercises
  - 10.10.2 Reading and Understanding Code
  - 10.10.3 Fill In the Code
  - 10.10.4 Identifying Errors in Code
  - 10.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
  - 10.10.6 Write a Short Program
  - 10.10.7 Programming Projects
  - 10.10.8 Technical Writing
  - 10.10.9 Group Project



Introduction

One of the most common ways to reuse a class is through inheritance. Inheritance helps us to organize related classes into **hierarchies**, or ordered levels of functionality. To set up a hierarchy, we begin by defining a class that contains methods and fields (instance variables and class variables) that are common to all classes in the hierarchy. Then we define new classes at the next lower level of the hierarchy, which inherit the behavior and fields of the original class. In the new classes, we define additional fields and more specific methods. The original class is called the **superclass**, and the new classes that inherit from the superclass are called **subclasses**. Some OOP developers call a superclass the **base class** and call a subclass the **derived class**.

As in life, a superclass (parent) can have multiple subclasses (children), and each subclass can be a superclass (parent) of other subclasses (children) and so on. Thus, a class can be both a subclass (child) and a superclass (parent). In contrast to life, however, Java subclasses inherit directly from only one superclass.

A subclass can add fields and methods, some of which may **override**, or hide, a field or method inherited from a superclass.

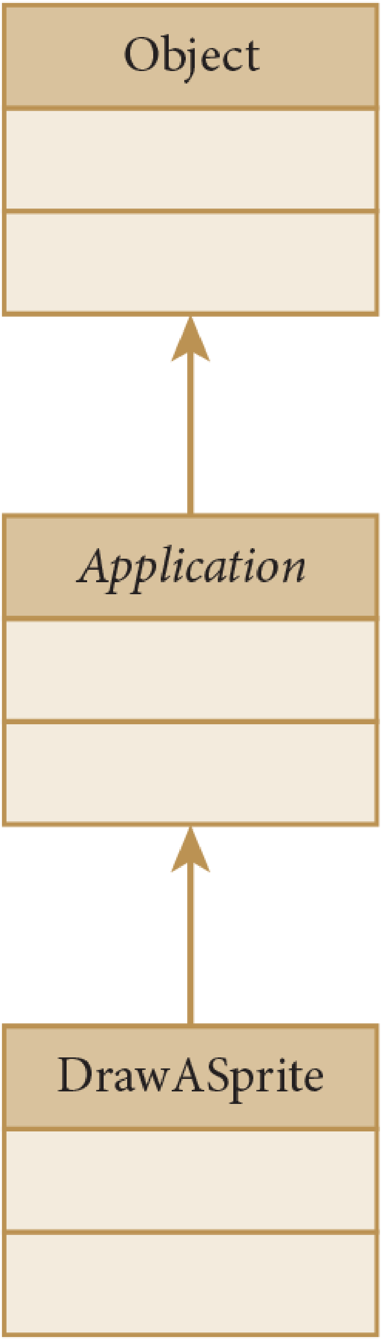
Let's look at an example. To represent a hierarchy of vehicle types, we define a *Vehicle* class as a superclass. We then define an *Automobile* class that inherits from *Vehicle*. We also define a *Truck* class, which also inherits from *Vehicle*. We further refine our classes by defining a *Pickup* class and a *TractorTrailer* class, both of which inherit from the *Truck* class. **Figure 10.1** depicts our hierarchy using a UML (Unified Modeling Language) diagram. Arrows pointing from a subclass to a superclass indicate that the subclass refers to the superclass for some of its methods and fields. The boxes below the class name are available for specifying instance variables and methods for each class. For simplicity, we leave those boxes blank. Later in the chapter, we will illustrate UML diagrams complete with fields and methods.

Figure 10.1 *Vehicle* Class Hierarchy

Description

The Java Class Library contains many class hierarchies. At the root of all Java class hierarchies is the *Object* class, the superclass for all classes. Thus, all classes inherit from the *Object* class.

The most important advantage to inheritance is that in a hierarchy of classes, we write the common code only once. After the common code has been tested, we can reuse it with confidence by inheriting it into the subclasses. And when that common code needs revision, we need to revise the code in only one place.


Object
Application
DrawASprite

10.1 Inheritance

The syntax for defining a subclass class that inherits from another class is to add an *extends* clause in the class header:

```
accessModifier class SubclassName extends SuperclassName
{
     // class definition
}
```

Description

The *extends* keyword specifies that the subclass inherits members of the superclass. That means that the subclass begins with a set of predefined methods and fields inherited from its hierarchy of superclasses.

For example, if we want to code a class named *DrawASprite*, we extend the *Application* class, and we use the following header:

```
public class DrawASprite extends Application
```

Description

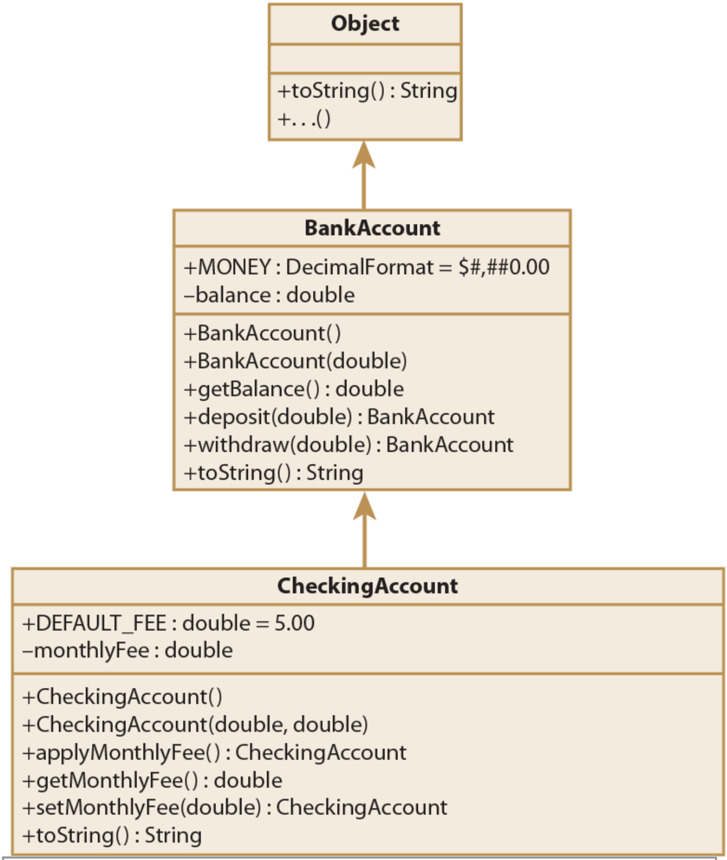
This means that the *DrawASprite* class inherits from the *Application* class.

The *DrawASprite* class hierarchy is shown in **Figure 10.2**. At the top of the hierarchy is the *Object* class. It defines 11 methods. The *Application* class extends the *Object* class, so it inherits those 11 methods. In turn, the *Application* class defines two fields and 10 additional methods. So when our *DrawASprite* class extends *Application*, the *DrawASprite* class inherits 21 methods and two fields. All along the hierarchy, subclasses inherit methods and fields. True, not every class has a use for all the inherited methods and fields, but they are available if needed, and the benefit is that we don’t need to write these methods or define these fields in our classes. Thus, we can build classes with a minimum of effort.

Figure 10.2 The *DrawASprite* Class Hierarchy

Description

As we can see from Figure 10.2, our *DrawASprite* class has two superclasses. The class that a subclass refers to in the *extends* clause of the class definition is called its **direct superclass**. Thus, *Application* is the direct superclass of *DrawASprite*. Similarly, the class that *extends* the superclass is called the **direct subclass** of the superclass, so *DrawASprite* is a direct subclass of the *Application* class. A class can have multiple direct subclasses, but only one direct superclass.


Object
+toString() : String
+. . .()
BankAccount
+MONEY : DecimalFormat = $#,##0.00
−balance : double
+BankAccount()
+BankAccount(double)
+getBalance() : double
+deposit(double) : BankAccount
+withdraw(double) : BankAccount
+toString() : String
CheckingAccount
+DEFAULT_FEE : double = 5.00
−monthlyFee : double
+CheckingAccount()
+CheckingAccount(double, double)
+applyMonthlyFee() : CheckingAccount
+getMonthlyFee() : double
+setMonthlyFee(double) : CheckingAccount
+toString() : String

```
New checking account: balance is $0.00
After depositing $350.75: balance is $350.75
After withdrawing $200.25: balance is $150.50
```

```
In BankAccount default constructor
In CheckingAccount default constructor
New checking account: balance is $0.00

In BankAccount overloaded constructor
In CheckingAccount overloaded constructor
New checking account: balance is $100.00
```

```
New checking account:
balance is $100.00; monthly fee is 7.5

After charging monthly fee:
balance is $92.50; monthly fee is 7.5
```

```
New checking account:
balance is $100.00; monthly fee is $7.50
```

10.2 Inheritance Design

We say that an “is a” relationship exists between a subclass and a superclass; that is, a subclass object “is a” superclass object. For example, we could define a student class hierarchy with a *Student* superclass and derive a *GraduateStudent* subclass. A graduate student “is a” student, but actually a special type of student. We could also define an employee class hierarchy with an *Employee* superclass and derive *Faculty* and *Staff* subclasses, because faculty and staff are both special types of employees.

To design classes for inheritance, our superclass should define fields and methods that will be common to all classes in the hierarchy. Each subclass will provide specialization by adding methods and fields. Where appropriate, subclasses can also provide new versions of inherited methods, which is called **overriding methods**.

SOFTWARE ENGINEERING TIP

The superclasses in a class hierarchy should contain fields and methods common to all subclasses. The subclasses should add specialized fields and methods.

Let’s build a bank account class hierarchy. We start by defining a generic *BankAccount* superclass. The *BankAccount* class will contain the fields and methods that are common to all bank accounts. Then we will define a *CheckingAccount* class that inherits from the *BankAccount* class. The *CheckingAccount* class will add instance variables and methods that specifically support checking accounts. Our class hierarchy is shown in the UML diagram in **Figure 10.3**. In this diagram, we display the instance variables in the box immediately below the class name and the methods in the next lower box. A “+” preceding a class member indicates that the member is *public*, while a “–” indicates that the member is *private*. Each method’s signature is given with each parameter’s type within parentheses and the return type following a colon. The *Object* class has more methods than we indicate on the UML diagram. However, the *toString* method is the only method of *Object* that we will deal with in this hierarchy, so we have omitted the other methods of *Object* on the diagram and indicate that other methods exist (+ . . . ( )).

Figure 10.3 The *BankAccount* Class Hierarchy

Description

10.2.1 Inherited Members of a Class

As shown in **Example 10.1**, our *BankAccount* class has two instance variables, the *balance*, which is a *double* (line 11), and a constant *DecimalFormat* object that we will use for formatting the *balance* as money (lines 9–10). We provide two constructors. The default constructor (lines 13–19) sets the *balance* instance variable to 0.0. The overloaded constructor (lines 21–27) takes a starting balance and passes that parameter to the *deposit* method (lines 37–48), which adds any non-negative amount to the *balance*. Otherwise, it leaves *balance* unchanged.

The *withdraw* method (lines 50–62) validates that the *amount* parameter is not less than 0.0 and is not greater than the *balance*. If *amount* is valid, the *withdraw* method subtracts *amount* from *balance*; otherwise, it leaves *balance* unchanged.

Example 10.1 *BankAccount* Class, Version 1

```java
/**   BankAccount class, version 1
*     Anderson, Franceschi
*     Represents a generic bank account
*/
import java.text.DecimalFormat;

public class BankAccount
{
  public final DecimalFormat MONEY
                  = new DecimalFormat( "$#,##0.00" );
  private double balance;

  /** default constructor
  *   sets balance to 0.0
  */
  public BankAccount( )
  {
    balance = 0.0;
  }

  /** overloaded constructor
  *   @param balance beginning balance
  */
  public BankAccount( double balance )
  {
    deposit( balance );
  }

  /** accessor for balance
  *   @return  current account balance
  */
  public double getBalance( )
  {
    return balance;
  }

  /** deposit amount to account
  *   @param amount  amount to deposit;
  *                  amount must be >= 0.0
  *   @return  a reference to this object
  */
  public BankAccount deposit( double amount )
  {
    if ( amount >= 0.0 )
      balance += amount;

    return this;
  }

  /** withdraw amount from account
  *    @param amount   amount to withdraw;
  *                    amount must be >= 0.0
  *                    amount must be <= balance
  *    @return a reference to this object
  */
  public BankAccount withdraw( double amount )
  {
    if ( amount >= 0.0 && amount <= balance )
      balance -= amount;

    return this;
  }

  /** toString
  *  @return  the balance formatted as money
  */
  @Override
  public String toString( )
  {
    return "balance is " + MONEY.format( balance );
  }
}
```

Description

Other methods of the *BankAccount* class include the *balance* accessor (lines 29–35) and the *toString* method (lines 64–71), which uses the *DecimalFormat* object, *MONEY*, to return the balance formatted as money. Note that we write *@Override* before the header of the *toString* method. We are replacing the version of *toString* that is in the *Object* class, that is, we are overriding the *Object* version of the *toString* method. To replace a method, we must write our method header exactly the same as the method we are overriding. By using the *@Override* annotation, we signal to the compiler that our intention is to override the method. If our method header is not coded correctly, the compiler will generate an error. Thus, it is a way to check our work.

Now we can derive our *CheckingAccount* subclass. **Example 10.2** shows Version 1 of our *CheckingAccount* class. For this initial version, we simply define the *CheckingAccount* class as extending *BankAccount* (line 5). The body of our class is empty for now, so we can demonstrate the fields and methods that a subclass inherits from its superclass.

Example 10.2 *CheckingAccount* Class, Version 1

```
1 /* CheckingAccount class, Version 1
2    Anderson, Franceschi
3 */
4
5 public class CheckingAccount extends BankAccount
6 { }
```

Description

When a class *extends* a superclass, all the *public* fields and methods of the superclass (excluding constructors) are inherited. That means that the *CheckingAccount* class inherits the *MONEY* instance variable and the *getBalance*, *deposit*, *withdraw*, and *toString* methods from the *BankAccount* class. An inherited field is directly accessible from the subclass, and an inherited method can be called by the other methods of the subclass. In addition, *public* inherited methods can be called by a client application using a subclass object reference.

Any fields and methods that are declared *private* are not inherited, and therefore are not directly accessible by the subclass. Nevertheless, the *private* fields are still part of the subclass object. Remember that a *CheckingAccount* object “is a” *BankAccount* object, so a *CheckingAccount* object has a *balance* instance variable. However, the *balance* is declared to be *private* in the *BankAccount* class, so the *CheckingAccount* methods cannot directly access the *balance*. The *CheckingAccount* methods must call the accessor and mutator methods of the *BankAccount* class to access or change the value of *balance*.

Calling methods to retrieve and change values of an instance variable may seem a little tedious, but it enforces encapsulation. Allowing the *CheckingAccount* class to set the value of *balance* directly would complicate the maintenance of the program. The *CheckingAccount* class would need to be responsible for maintaining a valid value for *balance*, which means that the *CheckingAccount* class would need to know all the validation rules for *balance* that the *BankAccount* class enforces. If these rules change, then the *CheckingAccount* class would also need to change. As long as the *BankAccount* class ensures the validity of *balance*, there is no reason for the *CheckingAccount* class to duplicate that code.

Java provides the ***protected*** access modifier so that fields and methods can be inherited by subclasses (like *public* fields and methods), while still being hidden from client classes (like *private* fields and methods). In addition, any class in the same package as the superclass can directly access a *protected* field, even if that class is not a subclass. Because more than one class can directly access a *protected* field, *protected* access compromises encapsulation and complicates the maintenance of a program. For that reason, we prefer to use *private*, rather than *protected*, for our instance variables. We will discuss the difference between *private* and *protected* in greater detail later in the chapter.

Table 10.1 summarizes the fields and methods that are inherited by a subclass. We will add to this table as we explain more about inheritance.

Inheritance Rules

Superclass Members	**Inherited by Subclass?**	**Directly Accessible by Subclass?**	**Directly Accessible by Client of Subclass?**
public fields	yes	yes, by using field name	yes
public methods	yes	yes, by calling method from other subclass methods	yes
protected fields	yes	yes, by using field name	no, must use accessors and mutators
protected methods	yes	yes, by calling method from subclass methods	no
private fields	no	no, must use accessors and mutators	no, must use accessors and mutators
private methods	no	no	no

Description

Example 10.3 shows a client for the *CheckingAccount* class. In line 9, we instantiate an object of the *CheckingAccount* class. After instantiation, the *c1* object has two fields (*balance* and *MONEY*), and it has inherited four methods (*getBalance*, *deposit*, *withdraw*, and *toString*) from *BankAccount*.

We illustrate this by using the *c1* object reference to call the *deposit* method in line 12 and the *withdraw* method in line 15, and to call the *toString* method implicitly in lines 13 and 16. **Figure 10.4** shows the output from this program.

Figure 10.4 Output from *CheckingAccountClient*, Version 1

Description

Example 10.3 *CheckingAccountClient*, Version 1

```
/* CheckingAccount Client, Version 1
   Anderson, Franceschi
*/

public class CheckingAccountClient
{
 public static void main( String [ ] args )
   {
   CheckingAccount c1 = new CheckingAccount( );
   System.out.println( "New checking account: " + c1 );

   c1.deposit( 350.75 );
   System.out.println( "\nAfter depositing $350.75: " + c1 );

   c1.withdraw( 200.25 );
   System.out.println( "\nAfter withdrawing $200.25: " + c1 );
 }
}
```

Description

10.2.2 Subclass Constructors

Although constructors are *public*, they are not inherited by subclasses. However, to initialize the *private* instance variables of the superclass, a subclass constructor can call a superclass constructor either implicitly or explicitly.

When a class extends another class, the default constructor of the subclass automatically calls the default constructor of the superclass. This is called **implicit** invocation. Athough we did not code any constructors in our *CheckingAccount* class in Example 10.2, we were able to instantiate a *CheckingAccount* object (with a 0.0 *balance*) because the Java compiler provided a default constructor for the *CheckingAccount* class, which implicitly called the default constructor of the *BankAccount* class.

To **explicitly** call the constructor of the direct superclass, the subclass constructor uses the following syntax:

```
super( argument list );
```

Description

Thus, if we want to instantiate a *CheckingAccount* object with a starting balance other than 0.0, we need to provide an overloaded constructor for the *CheckingAccount* class. That constructor will take the starting balance as a parameter and pass that starting balance to the overloaded constructor in the *BankAccount* class.

COMMON ERROR TRAP

In a constructor, the call to the direct superclass constructor, if used, must be the first statement.

This call to the direct superclass constructor, if used, must be the first statement in the subclass constructor. Otherwise, the following compiler error is generated:

```
call to super must be first statement in constructor
```

Description

Example 10.4 shows Version 2 of the *BankAccount* class, which, for simplicity and to help us focus on constructors, has only a default and overloaded constructor and the *toString* method. To illustrate the order in which the constructors execute, we print a message in each constructor (lines 21 and 33), indicating that it has been called.

Example 10.4 *BankAccount* Class, Version 2

```
/**   BankAccount class, Version 2
*     Constructors and toString method only
*     Anderson, Franceschi
*     Represents a generic bank account
*/

import java.text.DecimalFormat;

public class BankAccount
{
   public final DecimalFormat MONEY
                  = new DecimalFormat( "$#,##0.00" );
   private double balance;

   /** default constructor
   *   sets balance to 0.0
   */
   public BankAccount( )
   {
     balance = 0.0;
     System.out.println( "In BankAccount default constructor" );
   }

   /** overloaded constructor
   *   @param balance beginning balance
   */
   public BankAccount( double balance )
   {
     if ( balance >= 0.0 )
         this.balance = balance;
     else
         this.balance = 0.0;
     System.out.println( "In BankAccount overloaded constructor" );
   }

   /** toString
   *  @return the balance formatted as money
   */
   @Override
   public String toString( )
   {
     return "balance is " + MONEY.format( balance );
   }
}
```

Description

Example 10.5 shows Version 2 of the *CheckingAccount* class, which has both a default constructor and an overloaded constructor. Again, we have inserted messages (lines 13–14 and 24–25) to indicate when a constructor is called.

Example 10.5 *CheckingAccount* Class, Version 2

```
 1 /*  CheckingAccount class, Version 2
 2     Anderson, Franceschi
 3 */
 4
 5 public class CheckingAccount extends BankAccount
 6 {
 7    /** default constructor
 8    *   explicitly calls the BankAccount default constructor
 9    */
10    public CheckingAccount( )
11    {
12        super( ); // optional, call BankAccount constructor
13        System.out.println( "In CheckingAccount "
14                             + "default constructor" );
15    }
16
17    /** overloaded constructor
18    *   calls BankAccount overloaded constructor
19    *   @param balance starting balance
20    */
21    public CheckingAccount( double balance )
22    {
23        super( balance ); // call BankAccount constructor
24        System.out.println( "In CheckingAccount "
25                            + "overloaded constructor" );
26    }
27 }
```

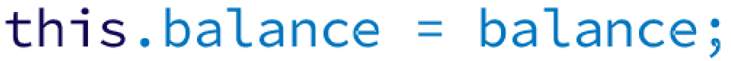
Description

In the *CheckingAccount* default constructor, we explicitly call the default constructor of the *BankAccount* class (line 12). This statement is optional; without it, the *BankAccount* default constructor is still called implicitly.

COMMON ERROR TRAP

An attempt by a subclass to directly access a *private* field or call a *private* method defined in a superclass will generate a compiler error. To set initial values for *private* variables, call the appropriate constructor of the direct superclass.

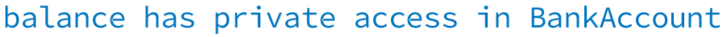
In the *CheckingAccount* overloaded constructor, we pass the *balance* parameter to the *BankAccount* constructor (line 23) to initialize the *balance* instance variable. Because the *balance* instance variable has *private* access in the *BankAccount* class, our *CheckingAccount* class cannot access the *balance* instance variable directly. If we attempted to initialize the *balance* directly using the following statement:



```
this.balance = balance;
```

Description

the compiler would generate the following error:



```
balance has private access in BankAccount
```

Description

We might be tempted to call the *deposit* method instead of *super* to initialize the balance. Although this would work in this case, we don't always know what operations a constructor is performing. Perhaps the constructor also assigns an account number, in which case, calling the *deposit* method would cause us to skip that step. Thus, it is always a good practice to call the constructor of the superclass so that the superclass data will be correctly initialized.

SOFTWARE ENGINEERING TIP

Overloaded constructors in a subclass should explicitly call the direct superclass constructor to initialize the fields in its superclasses.

Example 10.6 shows Version 2 of our *CheckingAccount* client. On line 10, we instantiate a *CheckingAccount* object using the default constructor and print the balance by implicitly calling the *toString* method on line 11. Then on line 14, we instantiate a second *CheckingAccount* object with a starting balance of $100.00. Again we verify the result by printing the balance (line 15).

Example 10.6 *CheckingAccountClient*, Version 2

```
/* CheckingAccount Client, Version 2
   Anderson, Franceschi
*/

public class CheckingAccountClient
{
   public static void main( String [ ] args )
   {
     // use default constructor
     CheckingAccount c1 = new CheckingAccount( );
     System.out.println( "New checking account: " + c1 + "\n" );

     // use overloaded constructor
     CheckingAccount c2 = new CheckingAccount( 100.00 );
     System.out.println( "New checking account: " + c2 );
   }
}
```

Description

Figure 10.5 shows the output from this program. As we can see, when we construct the *c1* object, the *BankAccount* default constructor runs. When it finishes, the *CheckingAccount* default constructor runs. Similarly, when we construct the *c2* object, the *BankAccount* overloaded constructor runs, then the *CheckingAccount* overloaded constructor runs.

Figure 10.5 Output from Example 10.6

Description

Table 10.2 summarizes the inheritance rules for constructors.

Inheritance Rules for Constructors

Superclass Members	Inherited by Subclass?	Directly Accessible by Subclass?	Directly Accessible by Client of Subclass Using a Subclass Reference?
constructors	no	yes, using `super( arg list )` in a subclass constructor	no

Description

10.2.3 Adding Specialization to the Subclass

At this point, our *CheckingAccount* class provides no more functionality than the *BankAccount* class. But our purpose for defining a *CheckingAccount* class was to provide support for a specialized type of bank account. To add specialization to our *CheckingAccount* subclass, we define new fields and methods. For example, we can define a *monthlyFee* instance variable, as well as an accessor and mutator method for the monthly fee and a method to charge the monthly fee to the account.

Example 10.7 shows Version 3 of the *CheckingAccount* class with the specialization added. This version *extends* the complete *BankAccount* class shown in Example 10.1. We added the *monthlyFee* instance variable on line 8, as well as a constant default value for the monthly fee (line 7). Our default constructor (lines 10–18) still calls the default constructor of the *BankAccount* class to initialize the *balance*, but it also initializes the *monthlyFee* to the default value.

Similarly, the overloaded constructor (lines 20–30) passes the *balance* parameter to the overloaded constructor of the *BankAccount* class and adds a *monthlyFee* parameter to accept an initial value for the *monthlyFee*, which it passes to the *setMonthlyFee* mutator method (lines 50–60).

The *applyMonthlyFee* method (lines 32–40), which charges the monthly fee to the checking account, calls the *withdraw* method inherited from the *BankAccount* class to access the *balance* instance variable, which is declared *private* in the *BankAccount* class.

Example 10.7 *CheckingAccount*, Version 3

```
/* CheckingAccount class, version 3
   Anderson, Franceschi
*/

public class CheckingAccount extends BankAccount
{
   public final double DEFAULT_FEE = 5.00;
   private double monthlyFee;

   /** default constructor
   *   explicitly calls the BankAccount default constructor
   *   sets monthlyFee to default value
   */
   public CheckingAccount( )
   {
       super( ); // optional
       monthlyFee = DEFAULT_FEE;
   }

   /** overloaded constructor
   *  calls BankAccount overloaded constructor
   *  @param  balance  starting balance
   *  @param  monthlyFee starting monthly fee
   */
   public CheckingAccount( double balance,
                           double monthlyFee )
   {
      super( balance ); // call BankAccount constructor
      setMonthlyFee( monthlyFee );
   }

   /** applyMonthlyFee method
   * charges the monthly fee to the account
   * @return a reference to this object
   */
   public CheckingAccount applyMonthlyFee( )
   {
     withdraw( monthlyFee );
     return this;
   }

   /** accessor method for monthlyFee
   *  @return  monthlyFee
   */
   public double getMonthlyFee( )
   {
     return monthlyFee;
   }

   /** mutator method for monthlyFee
   *  @param monthlyFee new value for monthlyFee
   *  @return a reference to this object
   */
   public CheckingAccount setMonthlyFee( double monthlyFee )
   {
     if ( monthlyFee >= 0.0 )
        this.monthlyFee = monthlyFee;

     return this;
   }
}
```

Description

Example 10.8 shows Version 3 of our client program, which instantiates a *CheckingAccount* object and charges the monthly fee. The output is shown in Figure 10.6.

Figure 10.6 Output from *CheckingAccountClient*, Version 3

Description

Example 10.8 *CheckingAccountClient*, Version 3

```java
/* CheckingAccount Client, Version 3
   Anderson, Franceschi
*/

public class CheckingAccountClient
{
   public static void main( String [ ] args )
   {
     CheckingAccount c3 = new CheckingAccount( 100.00, 7.50 );
     System.out.println( "New checking account:\n"
                          + c3.toString( )
                          + "; monthly fee is "
                          + c3.getMonthlyFee( ) );

     c3.applyMonthlyFee( ); // charge the fee to the account
     System.out.println( "\nAfter charging monthly fee:\n"
                          + c3.toString( )
                          + "; monthly fee is "
                          + c3.getMonthlyFee( ) );
   }
}
```

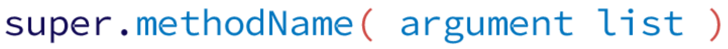
Description

10.2.4 Overriding Inherited Methods

When the methods our subclass inherits do not fulfill the functions we need, we can **override** the inherited methods by providing new versions of those methods.

To override an inherited method, we provide a new method with the same header as the inherited method; that is, the new method must have the same name, the same number and type of parameters, and the same return type. Overriding a method makes the inherited version of the method invisible to the client of the subclass. We say that the overridden method is hidden from the client. When the client calls the method using a subclass object reference, the subclass version of the method is invoked.

Methods in a subclass can still access the inherited version of the method by preceding the method call with the *super* object reference as in the following syntax:



```
super.methodName( argument list )
```

Description

In our *CheckingAccount* class, we inherited the *toString* method from the *BankAccount* class. But this method returns only the *balance*. In Example 10.8, we needed to call the *CheckingAccount* method *getMonthlyFee* to print the value of *monthlyFee*. Furthermore, as Figure 10.6 shows, the *balance* value is formatted and the *monthlyFee* value is not. Instead, the *toString* method in the *CheckingAccount* class should return formatted versions of both the *balance* and the *monthlyFee*. We can accomplish this by overriding the inherited *toString* method.

Example 10.9 shows Version 4 of the *CheckingAccount* class with the new *toString* method (lines 62–71). To format the *balance*, we call the *toString* method of the *BankAccount* class (line 69), then add the formatted value of *monthlyFee* to the *String* being returned. Again, we used the *@Override* annotation so that the compiler will alert us if we do not correctly code the header of the method. Notice that we didn’t need to instantiate a new *DecimalFormat* object in order to format the *monthlyFee* instance variable. Because the *MONEY* object is declared to be *public* in the *BankAccount* class, we inherited the *MONEY* object, so we can simply call the *format* method using the *MONEY* object reference. An advantage to making the *MONEY* object *public* is that both the balance and the monthly fee will be printed using the same formatting rules. Another advantage is that if we want to change the formatting for printing the data, we need to make only one change: We redefine the value of the *MONEY* constant in the *BankAccount* class.

Example 10.9 *CheckingAccount* Class, Version 4

```
/* CheckingAccount class, version 4
   Anderson, Franceschi
*/

public class CheckingAccount extends BankAccount
{
   public final double DEFAULT_FEE = 5.00;
   private double monthlyFee;

   /** default constructor
   *   explicitly calls the BankAccount default constructor
   *   sets monthlyFee to default value
   */
   public CheckingAccount( )
   {
       super( ); // call BankAccount constructor
       monthlyFee = DEFAULT_FEE;
   }

   /** overloaded constructor
   *  calls BankAccount overloaded constructor
   *  @param  balance  starting balance
   *  @param  monthlyFee starting monthly fee
   */
   public CheckingAccount( double balance,
                           double monthlyFee )
   {
      super( balance ); // call BankAccount constructor
      setMonthlyFee( monthlyFee );
   }

   /** applyMonthlyFee method
   * charges the monthly fee to the account
   * @return a reference to this object
   */
   public CheckingAccount applyMonthlyFee( )
   {
     withdraw( monthlyFee );
     return this;
   }

   /** accessor method for monthlyFee
   *  @return  monthlyFee
   */
   public double getMonthlyFee( )
   {
     return monthlyFee;
   }

   /** mutator method for monthlyFee
   *  @param monthlyFee new value for monthlyFee
   *  @return a reference to this object
   */
   public CheckingAccount setMonthlyFee( double monthlyFee )
   {
     if ( monthlyFee >= 0.0 )
        this.monthlyFee = monthlyFee;

     return this;
   }

   /* toString method
   *  @return String containing formatted balance and monthlyFee
   *    invokes superclass toString to format balance
   */
   @Override
   public String toString( )
   {
     return super.toString( )
            + "; monthly fee is " + MONEY.format( monthlyFee );
   }
}
```

Description

COMMON ERROR TRAP

Do not confuse overriding a method with overloading a method. A subclass overriding a method provides a new version of that method, which hides the superclass version. A class overloading a method adds a version of that method, which varies in the number and/or type of parameters.

Example 10.10 shows Version 4 of the *CheckingAccountClient* class. In this class, we again instantiate a *CheckingAccount* object with an initial balance of $100.00 and a monthly fee of $7.50 (line 9), then implicitly invoke the *toString* method to print the data of the object (line 10). This time, we invoke the *toString* method of the *CheckingAccount* class, which returns both the *balance* and *monthlyFee* values, formatted as money, as shown in **Figure 10.7**.

Figure 10.7 Output from *CheckingAccountClient*, Version 4

Description

Example 10.10 *CheckingAccountClient*, Version 4

```
 1 /* CheckingAccount Client, Version 4
 2    Anderson, Franceschi
 3 */
 4
 5 public class CheckingAccountClient
 6 {
 7    public static void main( String [ ] args )
 8    {
 9      CheckingAccount c4 = new CheckingAccount( 100.00, 7.50 );
10      System.out.println( "New checking account:\n" + c4 );
11    }
12 }
```

Description

Table 10.3 summarizes the inheritance rules for inherited methods that have been overridden.

Inheritance Rules for Overridden Methods

Superclass Members	Inherited by Subclass?	Directly Accessible by Subclass?	Directly Accessible by Client of Subclass Using a Subclass Reference?
public or *protected* inherited methods that have been overridden in the subclass	no	yes, using `super.methodName ( arg list )`	no

Description

When we override a method, the method signature must be identical to the inherited method. However, wherever the overridden method specifies a class as a parameter or return type, we can substitute a subclass for that parameter or return type.

This is possible because a subclass object is a superclass object, so a subclass object reference can be substituted for any superclass object reference. If two methods of a class have the same name but different signatures (that is, if the number, order, or type of parameters is different), then the method is *overloaded*, not *overridden*.

For example, if we were to write the *toString* method in the *CheckingAccount* class with the following header that specifies an *int* parameter:

```
public String toString( int a )
```

then our *toString* method would have a different signature from the *toString* method we inherited from the *BankAccount* class, which does not take any parameters. In this case, we are overloading the *toString* method, not overriding it. In other words, we are providing an additional version of the *toString* method. The inherited version is still visible and available to be called.

SOFTWARE ENGINEERING TIP

Methods that override inherited methods should explicitly call the direct superclass method whenever appropriate.

Table 10.4 illustrates the differences between overriding *public* methods and overloading *public* methods.

Overriding vs. Overloading Methods

	Method Names	**Argument Lists**	**Return Types**	**Directly Accessible by Subclass Client Using a Subclass Object Reference?**
Overriding a *public* Method	identical	identical	identical	only the subclass version can be called
Overloading a *public* Method	identical	different in number or type of parameters	identical	all versions of the overloaded method can be called

Description

with these end-of-chapter questions

10.10.1	Multiple Choice Exercises Questions 1, 2, 4, 8, 9
10.10.3	Fill In the Code Questions 21, 22, 23, 24
10.10.5	Debugging Area Questions 32, 34, 35
10.10.6	Write a Short Program Questions 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46
10.10.8	Technical Writing Question 56

```
New checking account:
balance is $100.00; monthly fee is $7.50

After withdrawing $95:
balance is $5.00; monthly fee is $7.50

Applying the monthly fee:
Warning: account is overdrawn!
balance is -$2.50; monthly fee is $7.50
```

10.3 The *protected* Access Modifier

We have seen that the subclass does not inherit constructors or *private* members of the superclass. However, the superclass constructors are still available to be called from the subclass and the *private* fields of the superclass are implemented as fields of the subclass.

Although *private* fields preserve encapsulation, there is additional processing overhead involved with calling methods. Whenever a method is called, the JVM saves the return address and makes copies of the arguments. Then when a value-returning method completes, the JVM makes a copy of the return value available to the caller. The *protected* access modifier was designed to avoid this processing overhead and to facilitate coding by allowing the subclass to access any *protected* field without calling its accessor or mutator method.

Be aware, however, that *protected* fields and methods also can be accessed directly by other classes in the same package, even if the classes are not within the same inheritance hierarchy.

To classes outside the package, a *protected* member of a class has the same restrictions as a *private* member. In other words, a class outside the package in which the *protected* member is declared may not call any *protected* methods and must access any *protected* fields through *public* accessor or mutator methods.

The *protected* access modifier has tradeoffs. As we mentioned, any fields declared as *protected* can be accessed directly by subclasses. Doing so, however, compromises encapsulation because multiple classes can set the value of a *protected* instance variable defined in another class.

SOFTWARE ENGINEERING TIP

Unless high performance is a critical requirement, avoid using the *protected* access modifier because doing so compromises encapsulation and complicates the maintenance of a program. Where possible, call superclass methods to change the values of *protected* instance variables.

Thus, maintaining classes that define or use *protected* members becomes more difficult. For example, we need to verify that any class that has access to the *protected* instance variable either does not set the variable's value, or if the class does change the value, that the new value is valid. Because of this added maintenance complexity, we recommend that *protected* access be used only when high performance is essential.

We also recommend that subclass methods avoid directly setting the value of a *protected* instance variable. Instead, wherever possible, call superclass methods when values of *protected* variables need to be changed.

To illustrate how *protected* access can be used in class hierarchies, let's look closely at our *CheckingAccount* class. We have been calling the *withdraw* method inherited from the *BankAccount* class to apply the monthly fee. However, the *withdraw* method leaves the *balance* unchanged if the withdrawal amount is greater than the balance. Thus, if the account does not have sufficient funds, the monthly fee is not charged. We would like the *CheckingAccount* class to be able to charge the monthly fee to the account and let the balance become negative.

To accomplish this, we declare the *balance* instance variable to be *protected* instead of *private*. This allows us to directly access *balance* inside the *applyMonthlyFee* method of the *CheckingAccount* class, because *balance* is now inherited by *CheckingAccount*.

Example 10.11 shows the *BankAccount* class, Version 3. The only change, compared to Version 1 (Example 10.1), is that the *balance* instance variable is declared as *protected*, rather than *private* (line 11).

Example 10.11 *BankAccount* Class, Version 3

```java
/**   BankAccount class, version 1
*     Anderson, Franceschi
*     Represents a generic bank account
*/
import java.text.DecimalFormat;

public class BankAccount
{
  public final DecimalFormat MONEY
                  = new DecimalFormat( "$#,##0.00" );
  protected double balance;

  /** default constructor
  *   sets balance to 0.0
  */
  public BankAccount( )
  {
    balance = 0.0;
  }

  /** overloaded constructor
  *   @param balance  beginning balance
  */
  public BankAccount( double balance )
  {
    deposit( balance );
  }

  /** accessor for balance
  *   @return  current account balance
  */
  public double getBalance( )
  {
    return balance;
  }

  /** deposit amount to account
  *   @param amount  amount to deposit;
  *                  amount must be >= 0.0
  *   @return  a reference to this object
  */
  public BankAccount deposit( double amount )
  {
    if ( amount >= 0.0 )
      balance += amount;

    return this;
  }

  /** withdraw amount from account
  *    @param amount   amount to withdraw;
  *                    amount must be >= 0.0
  *                    amount must be <= balance
  *    @return a reference to this object
  */
  public BankAccount withdraw( double amount )
  {
    if ( amount >= 0.0 && amount <= balance )
      balance -= amount;

    return this;
  }

  /** toString
  *  @return  the balance formatted as money
  */
  @Override
  public String toString( )
  {
    return "balance is " + MONEY.format( balance );
  }
}
```

Description

Example 10.12 shows Version 5 of the *CheckingAccount* class, which inherits from the *BankAccount* class in Example 10.11 that declares the *balance* as *protected*. The *CheckingAccount* class now inherits *balance*, and our *CheckingAccount* methods can access the *balance* variable directly. Nevertheless, in the default and overloaded constructors, we still call the superclass constructor to set the value of *balance* (lines 16 and 28). Otherwise, to avoid setting *balance* to an invalid initial value, we would need to know the validation rules for *balance* in *BankAccount* and unnecessarily duplicate that code.

Also, in the *toString* method (lines 63–72), we call the *toString* method of the *BankAccount* class. Again, we do this to be consistent with the superclass functionality, which formats the balance, and to avoid duplicating code.

In the *applyMonthlyFee* method (lines 32–41), however, we access *balance* directly. For this checking account, our bank will charge the monthly fee even if it results in a negative balance for the account, so we subtract *monthlyFee* from *balance*, which allows the balance to be negative. Notice that we change the value of *balance* directly instead of calling the *withdraw* method, which does not allow the balance to become negative.

Example 10.12 *CheckingAccount* Class, Version 5

```
/* CheckingAccount class, version 5
   Anderson, Franceschi
*/

public class CheckingAccount extends BankAccount
{
   public final double DEFAULT_FEE = 5.00;
   private double monthlyFee;

   /** default constructor
   *   explicitly calls the BankAccount default constructor
   *   set monthlyFee to default value
   */
   public CheckingAccount( )
   {
     super( );   // call BankAccount constructor
     monthlyFee = DEFAULT_FEE;
   }

   /** overloaded constructor
   *  calls BankAccount overloaded constructor
   *  @param  balance    starting balance
   *  @param  monthlyFee starting monthly fee
   */
   public CheckingAccount( double balance,
                           double monthlyFee )
   {
     super( balance );   // call BankAccount constructor
     setMonthlyFee( monthlyFee );
   }

   /** applyMonthlyFee method
   *   charges the monthly fee to the account
   *   @return a reference to this object
   */
   public CheckingAccount applyMonthlyFee( )
   {
     balance -= monthlyFee;

     return this;
   }

   /** accessor method for monthlyFee
   *  @return  monthlyFee
   */
   public double getMonthlyFee( )
   {
     return monthlyFee;
   }

   /** mutator method for monthlyFee
   *  @param monthlyFee new value for monthlyFee
   *  @return a reference to this object
   */
   public CheckingAccount setMonthlyFee( double monthlyFee
   {
     if ( monthlyFee >= 0.0 )
        this.monthlyFee = monthlyFee;

     return this;
   }

   /* toString method
   *  @return String containing formatted balance and month
   *    invokes superclass toString to format balance
   */
   @Override
   public String toString( )
   {
     return super.toString( )
            + "; monthly fee is " + MONEY.format( monthlyFe
   }
}
```

Description

Example 10.13 shows Version 5 of the *CheckingAccountClient* class. In this class, we again instantiate a *CheckingAccount* object with an initial balance of $100.00 and a monthly fee of $7.50 (line 9). We then call *withdraw* (line 12), so that the resulting balance is less than the monthly fee. Next we call the *applyMonthlyFee* method (line 16). We then check whether the balance is negative (line 17). If so, we output a warning message that the account is overdrawn (line 18). In any case, we output the new balance (line 19). The output of Example 10.13 is shown in **Figure 10.8**.

Figure 10.8 Output from *CheckingAccountClient*, Version 5

Description

Example 10.13 *CheckingAccountClient* Class, Version 5

```
 1 /*   CheckingAccount Client, Version 5
 2      Anderson, Franceschi
 3 */
 4
 5 public class CheckingAccountClient
 6 {
 7  public static void main( String [ ] args )
 8  {
 9     CheckingAccount c5 = new CheckingAccount( 100.00, 7.50 );
10     System.out.println( "New checking account:\n" + c5 );
11
12     c5.withdraw( 95 );
13     System.out.println( "\nAfter withdrawing $95:\n" + c5 );
14
15     System.out.println( "\nApplying the monthly fee:" );
16     c5.applyMonthlyFee( );
17     if ( c5.getBalance( ) < 0.0 )
18          System.out.println( "Warning: account is overdrawn!" );
19     System.out.println( c5 );
20 }
21}
```

Description

Table 10.5 compiles all the inheritance rules we have discussed.

Inheritance Rules

Superclass Members	Inherited by Subclass?	Directly Accessible by Subclass?	Directly Accessible by Client of Subclass?
public fields	yes	yes, by using field name	yes
public methods	yes	yes, by calling method from other subclass methods	yes, by calling method using a subclass object reference
protected fields	yes	yes, by using field name	no, must use accessors and mutators
protected methods	yes	yes, by calling method from other subclass methods	no
private fields	no	no, must use accessors and mutators	no, must use accessors and mutators
private methods	no	no	no
constructors	no	yes, using `super( arg list )` in a subclass constructor	no
public or *protected* inherited methods that have been overridden in the subclass	no	yes, using `super.methodName ( arg list )`	no

Description

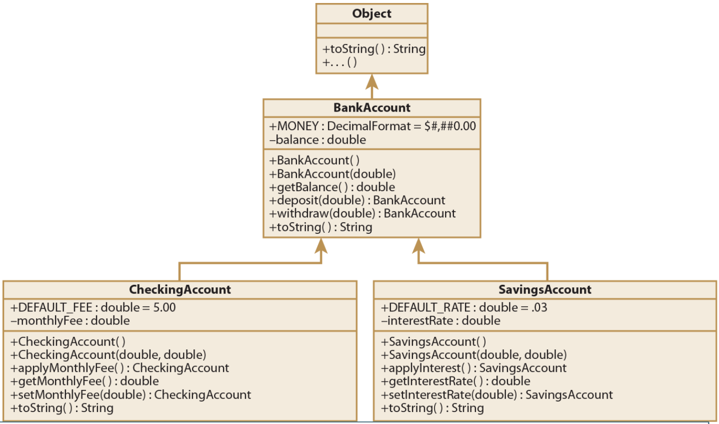
with these end-of-chapter questions

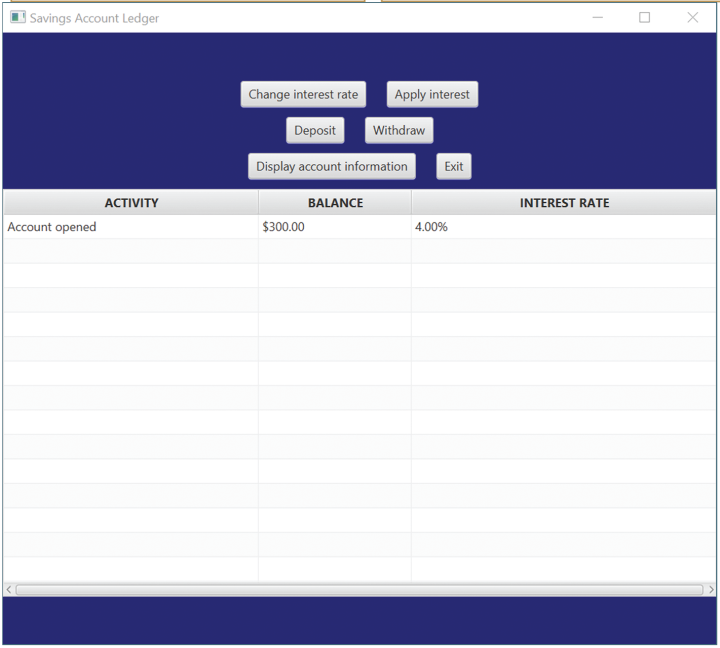
10.10.2	Reading and Understanding Code Questions 12, 13, 14, 15, 16, 17, 18, 19, 20

CODE IN ACTION

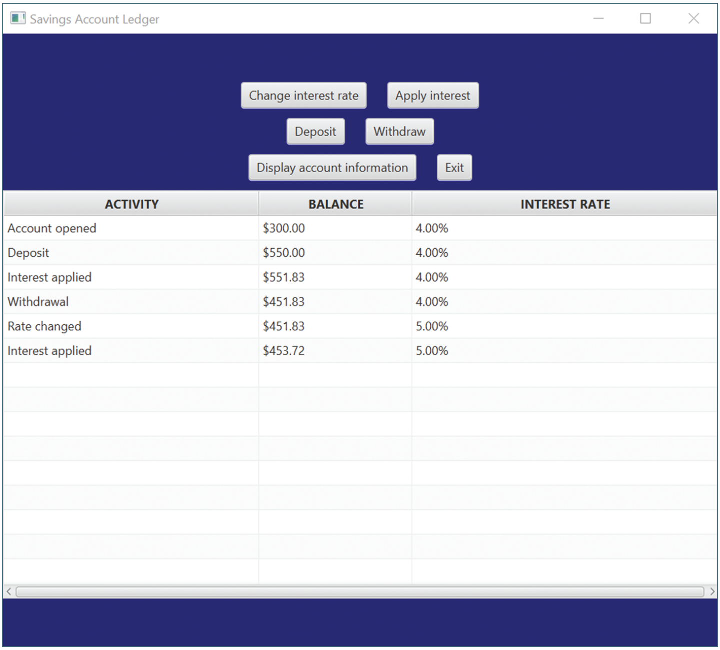
Within the online resources, you will find a movie with a step-by-step illustration of the use of inheritance in a program. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock


Object
+toString( ) : String
+. . . ( )
BankAccount
+MONEY : DecimalFormat = $#,##0.00
–balance : double
+BankAccount( )
+BankAccount(double)
+getBalance( ) : double
+deposit(double) : BankAccount
+withdraw(double) : BankAccount
+toString( ) : String
CheckingAccount
+DEFAULT_FEE : double = 5.00
–monthlyFee : double
+CheckingAccount( )
+CheckingAccount(double, double)
+applyMonthlyFee( ) : CheckingAccount
+getMonthlyFee( ) : double
+setMonthlyFee(double) : CheckingAccount
+toString( ) : String
SavingsAccount
+DEFAULT_RATE : double = .03
–interestRate : double
+SavingsAccount( )
+SavingsAccount(double, double)
+applyInterest( ) : SavingsAccount
+getInterestRate( ) : double
+setInterestRate(double) : SavingsAccount
+toString( ) : String


Savings Account Ledger
Change interest rate
Apply interest
Deposit
Withdraw
Display account information
Exit

ACTIVITY	BALANCE	INTEREST RATE
Account opened	$300.00	4.00%


Savings Account Ledger
Change interest rate
Apply interest
Deposit
Withdraw
Display account information
Exit

ACTIVITY	BALANCE	INTEREST RATE
Account opened	$300.00	4.00%
Deposit	$550.00	4.00%
Interest applied	$551.83	4.00%
Withdrawal	$451.83	4.00%
Rate changed	$451.83	5.00%
Interest applied	$453.72	5.00%

10.4 Programming Activity 1: Using Inheritance

For this Programming Activity, you will create the *SavingsAccount* class, which inherits directly from the *BankAccount* class. The *SavingsAccount* class is similar to the *CheckingAccount* class in that both classes inherit from *BankAccount*. **Figure 10.9** shows the resulting hierarchy.

Figure 10.9 Bank Account Hierarchy

Description

The *SavingsAccount* class inherits from the version of the *BankAccount* class in which the *balance* is declared to be *private*. The *SavingsAccount* subclass adds an annual *interestRate* instance variable, as well as supporting methods to access, change, and apply the interest rate to the account balance.

Instructions

Copy the source files in the Programming Activity 1 folder for this chapter to a folder on your computer. Load the *SavingsAccount.java* source file and search for five asterisks in a row (*****). This will position you to the six locations in the file where you will add code to complete the *SavingsAccount* class. The *SavingsAccount.java* file is shown in **Example 10.14**.

Example 10.14 *SavingsAccount.java*

```java
/* SavingsAccount class
*  Anderson, Franceschi
*/

import java.text.DecimalFormat;

// 1. ***** indicate that SavingsAccount inherits
//          from BankAccount
public class SavingsAccount
{
   public final double DEFAULT_RATE = .03;
   // 2. ****** define the private interestRate instance variable
   // interestRate, a double, represents an annual rate


   // 3 ***** write the default constructor
   /** default constructor
   *   explicitly calls the BankAccount default constructor
   *   set interestRate to default value DEFAULT_RATE
   *   print a message to System.out indicating that the
   *     constructor is called
   */


   // 4 ***** write the overloaded constructor
   /** overloaded constructor
   *   explicitly calls BankAccount overloaded constructor
   *   call setInterestRate method, passing interestRate
   *   print a message to System.out indicating that
   *     constructor is called
   *  @param  balance      starting balance
   *  @param  interestRate starting interest rate
   */


   // 5 ****** write this method:
   /** applyInterest method, no parameters.
   *   call the deposit method, passing a month's worth of interest
   *   remember that the interestRate instance variable is annual rate
   *
   *   @return a reference to this object
   */


   /** accessor method for interestRate
   *  @return  interestRate
   */
   public double getInterestRate( )
   {
     return interestRate;
   }

   /** mutator method for interestRate
   *  @param  interestRate new value for interestRate
   *          interestRate must be >= 0.0
   *            if not, do not change the value
   *          stores interestRate as input value / 100
   *          that is, 3.5 is stored as .035
   *  @return  a reference to this object
   */
   public SavingsAccount setInterestRate( double interestRate )
   {
     if ( interestRate >= 0.0 )
        this.interestRate = interestRate / 100;

     return this;
   }

   // 6 *****  write this method
   /* toString method
   *  @return a String containing formatted balance and interestRate
   *    invokes superclass toString to format balance
   *    formats interestRate as percent using a DecimalFormat object
   *    To create a DecimalFormat object for formatting percentages
   *    use this pattern in the constructor: "0.00%"
   */

}
```

Description

When you have completed the six tasks, load, compile, and run the Teller application (*Teller.java*), which you will use to test your *SavingsAccount* class. When the Teller application begins, you will be prompted with a dialog box for a starting balance. If you press “Enter” or the “OK” button without entering a balance, the Teller application will use the default constructor to instantiate a *SavingsAccount* object. If you enter a starting balance, the Teller application will prompt you for an interest rate and will instantiate a *SavingsAccount* object using the overloaded constructor. Once the *SavingsAccount* object has been instantiated, the Teller application will open the window shown in **Figure 10.10**, which provides buttons for calling the *SavingsAccount* methods to test your code.

Figure 10.10 The Teller Window

Description

Below the buttons is a ledger that displays the current state of the savings account. As you click on the various buttons, the ledger will display the operation performed and the values of the balance and the interest rate when that operation is complete.

The operations performed by each button are already coded for you and are the following:

- *Change* interest rate—prompts for a new interest rate and calls your *setInterestRate* method
- *Apply interest*—calls your *applyInterest* method
- *Deposit*—prompts for the deposit amount and calls the *deposit* method inherited from *BankAccount*
- *Withdraw*—prompts for the withdrawal amount and calls the *withdraw* method inherited from *BankAccount*
- *Display account information*—calls your *toString* method and displays the result in a dialog box
- *Exit*—exits the program

Figure 10.11 shows the Teller window after several operations have been performed.

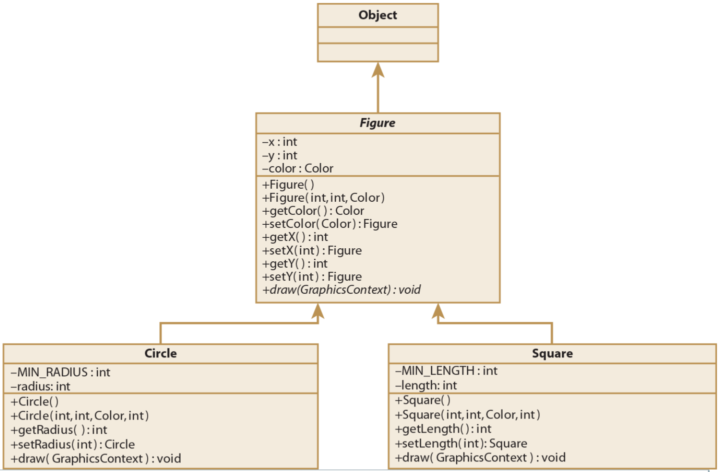
Figure 10.11 Sample Teller Window After Performing Several

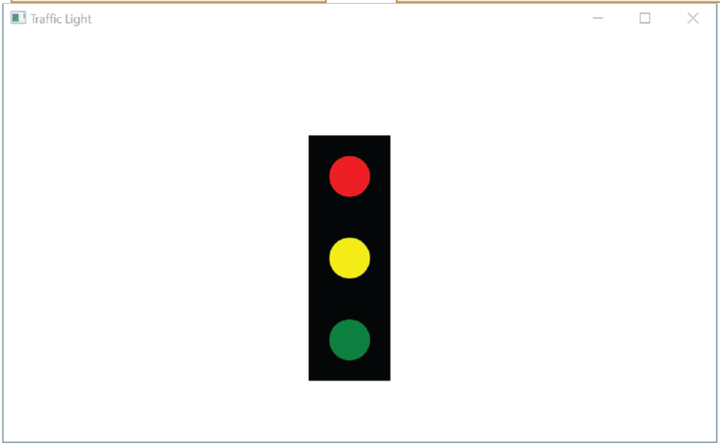
Operations

Description

DISCUSSION QUESTIONS

1. Explain why the Teller application can call the *withdraw* and *deposit* methods using a *SavingsAccount* object reference, even though you did not define these methods in the *SavingsAccount* class.
2. **Explain why your** *applyInterest* **method in the** *SavingsAccount* **class needs to call the** *deposit* **method of the** *BankAccount* **class.**


Object
Figure
−x : int
−y : int
−color : Color
+Figure( )
+Figure(int, int, Color)
+getColor( ) : Color
+setColor(Color) : Figure
+getX( ) : int
+setX(int) : Figure
+getY( ) : int
+setY(int) : Figure
+draw(GraphicsContext) : void
Circle
−MIN_RADIUS : int
−radius: int
+Circle( )
+Circle(int, int, Color, int)
+getRadius( ) : int
+setRadius(int) : Circle
+draw(GraphicsContext) : void
Square
−MIN_LENGTH : int
−length: int
+Square( )
+Square(int, int, Color, int)
+getLength( ) : int
+setLength(int): Square
+draw(GraphicsContext) : void


Traffic Light

10.5 *Abstract* Classes and Methods

In our Bank Account hierarchy, we could instantiate *BankAccount* objects, *CheckingAccount* objects, and *SavingsAccount* objects. In some situations, however, we will design a class hierarchy where one or more classes at the top of the hierarchy are not intended to be instantiated. Rather, they typically specify patterns for methods that subclasses in the hierarchy must implement. Often, the superclasses do not implement these methods. In these situations, we do not intend that these superclasses will be used to instantiate objects, and we define the superclasses as ***abstract***.

An ***abstract* class** is a class that is intended to be extended, rather than instantiated. Usually, an *abstract* class contains at least one ***abstract* method**, that is, a method that specifies an API that subclasses should implement, but does not provide an implementation for the method.

An *abstract* class cannot be used to instantiate objects. An *abstract* class can be extended, however, so that its subclasses can complete the implementation of the *abstract* methods and the subclasses can be instantiated.

A class is declared to be *abstract* by including the *abstract* keyword in the class header, as shown in the following syntax:

```
accessModifier abstract class ClassName
```

Description

An *abstract* method is defined by including the *abstract* keyword in the method header and by using a semicolon to indicate that there is no code for the method, as shown in the following syntax:

```
accessModifier abstract returnType methodName( argument list );
```

Description

Note that we do not include opening and closing curly braces for the method body—just a semicolon to indicate that the *abstract* method does not have a body.

Java imposes a few restrictions on *abstract* methods: constructors cannot be defined as *abstract*; and an *abstract* method cannot be declared as *private* or *static*.

For example, to draw figures, we can set up the hierarchy shown in **Figure 10.12**. The root superclass under *Object* is the *abstract Figure* class, and we derive two **concrete** (non-*abstract*) subclasses: *Circle* and *Square*. In the UML diagram, the name of the *Figure* class is set in italics to indicate that it is an *abstract* class.

Figure 10.12 The *Figure* Hierarchy

Description

All figures will have an (*x, y*) coordinate and a color, so the *Figure* class defines three fields: two *ints*, *x* and *y*, and a *Color* object named *color*.

We want all classes in the hierarchy to provide a *draw* method to render the figure; however, the *Figure* class has nothing but a point to draw, so its *draw* method has nothing to do. Thus, we do not provide an implementation of the *draw* method in the *Figure* class; instead, we define the *draw* method as an *abstract* method. Like the *abstract* class name, we indicate that the *draw* method is *abstract* by setting it in italics in the UML.

Let's look at the code for the *Figure* hierarchy in detail. **Example 10.15** shows the *abstract Figure* class. We define the class as *abstract* in the class header (line 7). The constructors (lines 13–22 and lines 24–34) initialize the *x* and *y* values and instantiate the *Color* object that all figures will have in common. The *Figure* class also provides accessor and mutator methods for its instance variables. The *abstract draw* method (lines 94–97) provides the API for the *draw* method, but no implementation—just a semicolon. The *Circle* and *Square* subclasses of the *Figure* class will provide appropriate implementations of the *draw* method.

Example 10.15 The *abstract Figure* Class

```java
/** abstract Figure superclass for drawing shapes
*  Anderson, Franceschi
*/
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;

public abstract class Figure
{
  private int x;
  private int y;
  private Color color;

  /** default constructor
  *   sets x and y to 0
  *   sets color to black
  */
  public Figure( )
  {
    x = 0;
    y = 0;
    color = Color.BLACK;
  }

  /** overloaded constructor
  *   @param  x      starting x coordinate for figure
  *   @param  y      starting y coordinate for figure
  *   @param  color  figure color
  */
  public Figure( int x, int y, Color color )
  {
    this.x = x;
    this.y = y;
    this.color = color;
  }

  /** accessor method for color
  *   @return current figure color
  */
  public Color getColor( )
  {

    return color;
  }

  /** mutator method for color
  *   @param color  new color for figure
  *   @return a reference  to this object
  */
  public Figure setColor( Color color )
  {
    this.color = color;

    return this;
  }

  /** accessor method for x
  *   @return current x value
  */
  public int getX( )
  {
    return x;
  }

  /** mutator method for x
  *   @param x  new value for x
  *   @return a reference to this object
  */
  public Figure setX( int x )
  {
    this.x = x;

    return this;
  }

  /** accessor method for y
  *   @return current y value
  */
  public int getY( )
  {
    return y;
  }

  /** mutator method for y
  *   @param y new y value
  *   @return a reference to this object
  */
  public Figure setY( int y )
  {
     this.y = y;

     return this;
  }

  /** abstract draw method
  *   @param gc GraphicsContext for drawing figure
  */
  public abstract void draw( GraphicsContext gc );
}
```

Description

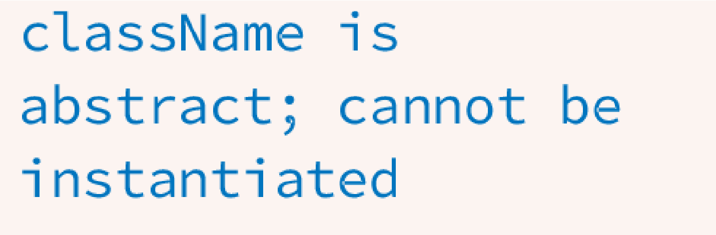
COMMON ERROR TRAP

Do not include opening and closing curly braces in the definition of an *abstract* method. Including them would mean that the method is implemented, but does nothing. Instead, indicate an unimplemented method by using only a semicolon.

When a subclass inherits from an *abstract* class, it can provide implementations for any, all, or none of the *abstract* methods. If the subclass does not completely implement all the *abstract* methods of the superclass, then the subclass must also be declared *abstract*. If, however, the subclass implements all the *abstract* methods in the superclass, and the subclass is not declared *abstract*, then the class is not *abstract* and we can instantiate objects of that subclass.

COMMON ERROR TRAP

Attempting to instantiate an object of an *abstract* class will generate the following compiler error:



```
className is
abstract; cannot be
instantiated
```

Description

where *className* is the name of the *abstract* class.

Example 10.16 shows the *Circle* class, which inherits from the *Figure* class and adds a *radius* instance variable, as well as a constant for the minimum radius. In the overloaded constructor, we pass the *x, y,* and *color* parameters to the constructor of the *Figure* class (line 35). On lines 61–71, the *Circle* class implements the *draw* method. We get the (*x, y*) coordinate and the color for the circle by calling the accessor methods of the *Figure* class because the *x, y*, and *color* instance variables are declared *private*.

Example 10.16 The *Circle* Class

```
/* Circle class
*  inherits from abstract Figure class
*  Anderson, Franceschi
*/

import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;

public class Circle extends Figure
{
   private final int MIN_RADIUS = 15;
   private int radius;

   /** default constructor
   *   calls default constructor of Figure class
   *   sets radius to minimum value
   */
   public Circle( )
   {
      super( );
      radius = MIN_RADIUS;
   }

   /** overloaded constructor
   *   sends x, y, color parameters to Figure constructor
   *   sends radius to setRadius method
   *   @param x      starting x coordinate
   *   @param y      starting y coordinate
   *   @param color  color for circle
   *   @param radius radius of circle
   */
   public Circle( int x, int y, Color color,
                             int radius )
   {
      super( x, y, color );
      setRadius( radius );
   }

   /** mutator method for radius
   *   @param radius  new value for radius
   *   @return a reference to this object
   */
   public Circle setRadius( int radius )
   {
      if ( radius > MIN_RADIUS )
         this.radius = radius;
      else
         this.radius = MIN_RADIUS;

      return this;
   }

   /** accessor method for radius
   *   @return radius
   */
   public int getRadius( )
   {
      return radius;
   }

   /** draw method
   *  sets color and draws a circle
   *  @param gc  GraphicsContext for drawing the circle
   */
   @Override
   public void draw( GraphicsContext gc )
   {
       gc.setFill( getColor( ) );
       gc.fillOval ( getX( ), getY( ),
                     radius * 2, radius * 2 );
   }
}
```

Description

Similarly, **Example 10.17** shows the *Square* class, which also inherits from the *Figure* class. The *Square* class adds a *length* instance variable, as well as a constant for the minimum length, and uses code similar to the *Circle* class to call the constructors of the *Figure* class (lines 20 and 35) and to implement its own version of the *draw* method (lines 61–71).

Example 10.17 The *Square* Class

```
/* Square class
*  inherits from abstract Figure class
*  Anderson, Franceschi
*/

import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;

public class Square extends Figure
{
   private final int MIN_LENGTH = 10;
   private int length;

   /** default constructor
   *   calls default constructor of Figure class
   *   sets length to minimum value
   */
   public Square( )
   {
     super( );
     length = MIN_LENGTH;
   }

   /** overloaded constructor
   *   sends x, y, and color parameters to Figure constructor
   *   sends length to setLength method
   *   @param x      starting x coordinate
   *   @param y      starting y coordinate
   *   @param color  color for square
   *   @param length length of square
   */
   public Square( int x, int y, Color color,
                  int length )
   {
      super( x, y, color );
      setLength( length );
   }

   /** mutator method for length
   *   @param length  new value for length
   *   @return a reference to this object
   */
   public Square setLength( int length )
   {
      if ( length > MIN_LENGTH )
         this.length = length;
      else
         this.length = MIN_LENGTH;

      return this;
   }

   /** accessor method for length
   *   @return length
   */
   public int getLength( )
   {
      return length;
   }

   /** draw method
   *   sets color and draws a square
   *   @param gc  GraphicsContext for drawing the square
   */
   @Override
   public void draw( GraphicsContext gc )
   {
      gc.setFill( getColor( ) );
      gc.fillRect( getX( ), getY( ),
                   length, length );
   }
}
```

Description

Because we want to instantiate *Circle* and *Square* objects, we do not declare these classes *abstract* and they are forced to implement the *draw* method. **Example 10.18** shows a graphical application, *TrafficLight*, which paints a traffic light, shown in **Figure 10.13**. On lines 13 and 14, we declare two *ArrayLists*, one to hold *Circle* objects and one to hold *Square* objects. In the *start* method, we instantiate both *ArrayLists* and add three *Square* objects to *squaresList* and three *Circle* objects to *circlesList* (lines 26–40). Then, we create the traffic light by calling the *draw* methods for all the *Squares* in the *squaresList* (lines 42–43), then calling the *draw* method for all the *Circles* in the *circlesList* (lines 45–46).

Figure 10.13 The *TrafficLight* Application

Example 10.18 The *TrafficLight* Application

```java
/* Figure Hierarchy Client
 * Anderson, Franceschi
*/
import javafx.application.Application;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

import java.util.ArrayList;

public class TrafficLight extends Application
{
   private ArrayList<Square> squaresList;
   private ArrayList<Circle> circlesList;

   @Override
   public void start( Stage stage )
   {
     GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
                           stage, "Traffic Light", 700, 400 );

     final int SQUARE_SIZE = 80, CIRCLE_RADIUS = 20;
     final int SQUARE_X = 300, CIRCLE_X = SQUARE_X + 20;
     final int SQUARE_Y = 100, CIRCLE_Y = SQUARE_Y + 20;

     squaresList = new ArrayList<Square>( );
     squaresList.add( new Square( SQUARE_X, SQUARE_Y,
                                  Color.BLACK, SQUARE_SIZE ) );
     squaresList.add( new Square( SQUARE_X, SQUARE_Y + SQUARE_SI
                                  Color.BLACK, SQUARE_SIZE ) );
     squaresList.add( new Square( SQUARE_X, SQUARE_Y + ( SQUARE_
                                  Color.BLACK, SQUARE_SIZE ) );

     circlesList = new ArrayList<Circle>( );
     circlesList.add( new Circle( CIRCLE_X, CIRCLE_Y,
                                  Color.RED, CIRCLE_RADIUS ) );
     circlesList.add( new Circle( CIRCLE_X, CIRCLE_Y + SQUARE_SI
                                  Color.YELLOW, CIRCLE_RADIUS )
     circlesList.add( new Circle( CIRCLE_X, CIRCLE_Y + ( SQUARE_
                                  Color.GREEN, CIRCLE_RADIUS ) )

     for ( Square s : squaresList )
       s.draw( gc );

     for ( Circle c : circlesList )
       c.draw( gc );
   }

   public static void main( String [ ] args )
   {
     launch( args );
   }
}
```

Description

Java’s restrictions on declaring and using *abstract* classes and methods are summarized in Table 10.6.

Restrictions for *abstract* Classes and *abstract* Methods Within Classes

abstract classes	Classes must be declared *abstract* if the class contains any *abstract* methods.
	abstract classes can be extended.
	abstract classes cannot be used to instantiate objects.
abstract methods within classes	*abstract* methods cannot be declared within a non-*abstract* class.
	An *abstract* method must consist of a method header followed by a semicolon.
	abstract methods cannot be declared as *private* or *static*.
	A constructor cannot be declared *abstract*.

Description

10.6 Polymorphism

An important concept in inheritance is that an object of a class is also an object of any of its superclasses. That concept, “is a”, is the basis for an important OOP feature, called **polymorphism**, which simplifies the processing of various objects in the same class hierarchy. The word *polymorphism*, which is derived from the word fragment *poly* and the word *morpho* in the Greek language, literally means “multiple forms.”

Polymorphism allows us to use the same method call for any object in the hierarchy. We make the method call using an object reference of the superclass. At run time, the JVM determines to which class in the hierarchy the object actually belongs and calls the version of the method implemented for that class.

To use polymorphism in our application, the following conditions must be true:

- The classes are in the same hierarchy.
- The subclasses override the same method.
- A subclass object reference is assigned to a superclass object reference (that is, a subclass object is referenced by a superclass reference).
- The superclass object reference is used to call the method.

For example, we can take advantage of polymorphism in our traffic light graphical application by calling the *draw* method for either a *Circle* or *Square* object using a *Figure* object reference. Although we cannot instantiate an object from an *abstract* class, Java allows us to define object references of an *abstract* class.

Example 10.19 shows the rewritten traffic light graphical application. Instead of using separate *ArrayLists* for *Circle* and *Square* objects, we can declare and instantiate only one *ArrayList* of *Figure* references (lines 13 and 25). As each *Circle* and *Square* object is instantiated, we add its object reference to the *ArrayList* of *Figure* references (lines 27–39).

This greatly simplifies drawing the traffic light because we step through just one *ArrayList*, *figuresList*, calling the *draw* method for each element (lines 41-42). For the method call, it doesn't matter whether the object reference in *figuresList* is a *Circle* or *Square* reference. We just call the *draw* method using that reference. At run time, the JVM determines whether the object is a *Circle* or a *Square* and calls the appropriate *draw* method for the object type. Because the *ArrayList* is composed of *Figure* references, any element can be either a *Circle* or a *Square*—because a *Circle* and a *Square* are both *Figures*. The output of this application is identical to that of Example 10.18, as shown in Figure 10.13.

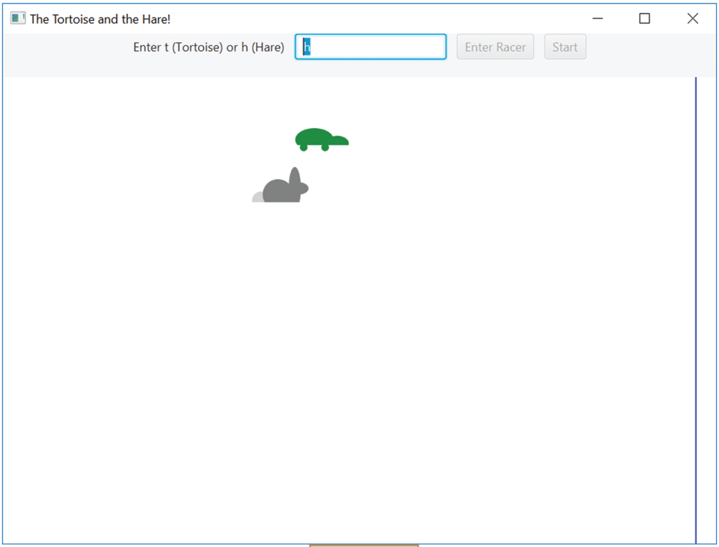
Example 10.19 Traffic Light Application Using Polymorphism

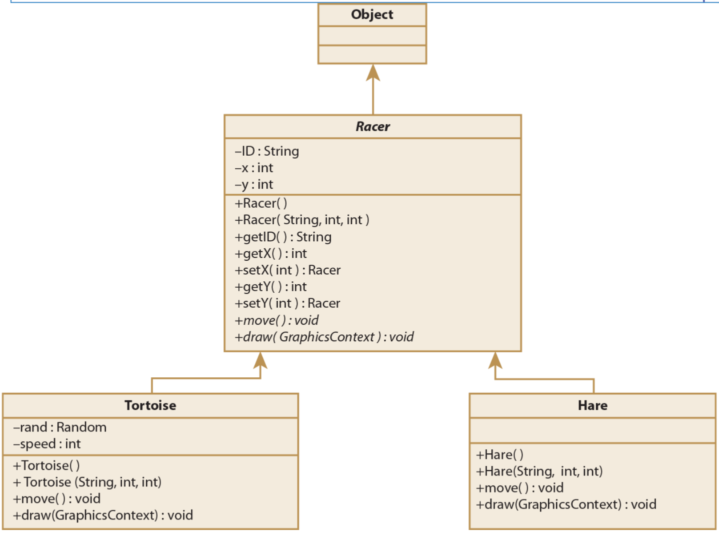
```
/* Figure hierarchy Client
 * Anderson, Franceschi
*/
import javafx.application.Application;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

import java.util.ArrayList;

public class TrafficLightPolymorphism extends Application
{
   private ArrayList<Figure> figuresList;

   @Override
   public void start( Stage stage )
   {
     GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
                          stage, "Traffic Light", 700, 400 );

     final int SQUARE_SIZE = 80, CIRCLE_RADIUS = 20;
     final int SQUARE_X = 300, CIRCLE_X = SQUARE_X + 20;
     final int SQUARE_Y = 100, CIRCLE_Y = SQUARE_Y + 20;

     figuresList = new ArrayList<Figure>( );

     figuresList.add( new Square( SQUARE_X, SQUARE_Y,
                                  Color.BLACK, SQUARE_SIZE ) );
     figuresList.add( new Square( SQUARE_X, SQUARE_Y + SQUARE_SIZE,
                                  Color.BLACK, SQUARE_SIZE ) );
     figuresList.add( new Square( SQUARE_X, SQUARE_Y + ( SQUARE_SIZE * 2 ),
                                  Color.BLACK, SQUARE_SIZE ) );

     figuresList.add( new Circle( CIRCLE_X, CIRCLE_Y,
                                  Color.RED, CIRCLE_RADIUS ) );
     figuresList.add( new Circle( CIRCLE_X, CIRCLE_Y + SQUARE_SIZE,
                                  Color.YELLOW, CIRCLE_RADIUS ) );
     figuresList.add( new Circle( CIRCLE_X, CIRCLE_Y + ( SQUARE_SIZE * 2 ),
                                  Color.GREEN, CIRCLE_RADIUS ) );

     for ( Figure f : figuresList )
        f.draw( gc );

   }

   public static void main( String [ ] args )
   {
     launch( args );
   }
}
```

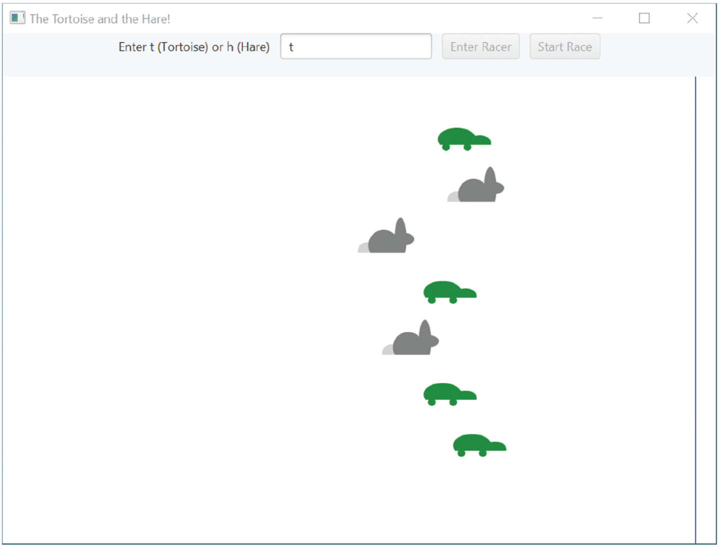
Description

with these end-of-chapter questions

10.10.1	Multiple Choice Exercises Questions 7,10,11
10.10.4	Identifying Errors in Code Question 31


The Tortoise and the Hare!
Enter t (Tortoise) or h (Hare)
h
Enter Racer
Start


Object
Racer
–ID : String
–x : int
–y : int
+Racer( )
+Racer(String, int, int)
+getID( ) : String
+getX( ) : int
+setX(int) : Racer
+getY( ) : int
+setY(int) : Racer
+move( ) : void
+draw(GraphicsContext) : void
Tortoise
–rand : Random
–speed : int
+Tortoise( )
+ Tortoise (String, int, int)
+move( ) : void
+draw(GraphicsContext) : void
Hare
+Hare( )
+Hare(String, int, int)
+move( ) : void
+draw(GraphicsContext) : void


The Tortoise and the Hare!

Enter t (Tortoise) or h (Hare) t

10.7 Programming Activity 2: Using Polymorphism

In this Programming Activity, you will complete the implementation of the Tortoise and the Hare race. The Tortoise runs a slow and steady race, while the Hare runs in spurts with rests in between. **Figure 10.14** shows a sample run of the race. In this figure, we show only one tortoise and one hare; however, using polymorphism we can easily run the race with any number and combination of tortoises and hares.

Figure 10.14 A Sample Run of the Tortoise and the Hare Race

Description

The class hierarchy for this Programming Activity is shown in **Figure 10.15**.

Figure 10.15 *Racer* Hierarchy

Description

The code for the *Racer* class, which is the superclass of the *Tortoise* and *Hare* classes, is shown in **Example 10.20**. The

Racer class has three instance variables (lines 10–12): a *String* *ID*, which identifies the type of racer; and *x* and *y* positions, both of which are *ints*. The class has the usual constructors, as well as accessor and mutator methods for the *x* and *y* positions and ID. These instance variables and methods are common to all racers, so we define them in the *Racer* class. Individual racers, however, will differ in the way they move and in the way they are drawn. Thus, in line 8, we declare the *Racer* class to be *abstract*, and in lines 78–85, we define two *abstract* methods, *move* and *draw*. Classes that inherit from the *Racer* class will need to provide implementations of these two methods (or be declared *abstract* as well).

Example 10.20 The *abstract Racer* Class

```
/**  Racer class
*    Abstract class intended for racer hierarchy
*    Anderson, Franceschi
*/

import javafx.scene.canvas.GraphicsContext;

public abstract class Racer
{
  private String ID;  // racer ID
  private int x;      // x position
  private int y;      // y position

  /** default constructor
  *    Sets ID to blank
  */
  public Racer( )
  {
    ID = "";
  }

  /** constructor
  *    @param ID   racer ID
  *    @param x    x position
  *    @param y    y position
  */
  public Racer( String ID, int x, int y )
  {
    this.ID = ID;
    this.x = x;
    this.y = y;
  }

  /** accessor for ID
  *   @return  ID
  */
  public String getID( )
  {
    return ID;
  }

  /** accessor for x
  *   @return  current x value
  */
  public int getX( )
  {
    return x;
  }

  /** accessor for y
  *   @return  current y value
  */
  public int getY( )
  {
    return y;
  }

  /** mutator for x
  *   @param  x new value for x
  *   @return a reference to this object
  */
  public Racer setX( int x )
  {
    this.x = x;
    return this;
  }

  /** mutator for y
  *   @param  y new value for y
  *   @return a reference to this object
  */
  public Racer setY( int y )
  {
    this.y = y;
    return this;
  }

  /** abstract method for Racer's move
  */
  public abstract void move( );

  /** abstract method for drawing Racer
  *   @param   gc    GraphicsContext
  */
  public abstract void draw( GraphicsContext gc );
}
```

Description

The *Tortoise* and *Hare* classes inherit from the *Racer* class. Their only job is to pass constructor arguments to the *Racer* class and implement the *draw* and *move* methods. For this Programming Activity, we have provided the *Tortoise* and *Hare* classes with the *draw* and *move* methods already written.

Your job is to add *Tortoise* and *Hare* objects to an *ArrayList* of *Racer* objects, as specified by the user. Then you will add code to run the race by stepping through the *ArrayList*, calling *move* and *draw* for each *Racer* object.

Instructions

Copy the source files in the Programming Activity folder for this chapter to a folder on your computer. Open the *PolymorphismController.java* file.

1. Write the code to determine which racers will run the race. Search for five asterisks in a row (*****). This will position you inside the *prepareToRace* method.

```
/** prepareToRace method
*    @param input     racer type entered by the user
*                     racer types are 't' or 'T' for Tortoise,
*                                     'h' or 'H' for Hare
*/
private void prepareToRace( char input )
{
   final int START_LINE = 60;     // x position of start of race
   final int RACER_SPACE = 50;    // spacing between racers

   /** 1. ***** Student writes this switch statement
   *  input parameter contains the racer type
   *      entered by the user
   *  If input is 'T' or 't',
   *     add a Tortoise object to the ArrayList named racerList,
   *            which is an instance variable of this class
   *  The API of the Tortoise constructor is:
   *          Tortoise( String ID, int x, int y )
   *   a sample call to the constructor is
   *          new Tortoise( "Tortoise", START_LINE, yPos )
   *          where START_LINE is a constant local variable
   *            representing the starting x position for the race
   *          and yPos is an instance variable representing
   *             the racer's y position
   *
   *  If input is 'H' or 'h',
   *      add a Hare object to the ArrayList named racerList
   *  The API of the Hare constructor is:
   *          Hare( String ID, int x, int y )
   *      a sample call to the constructor is
   *          new Hare( "Hare", START_LINE, yPos )
   *          where START_LINE is a constant local variable
   *            representing the starting x position for the race
   *          and yPos is an instance variable representing
   *             the racer's y position
   *
   *  After adding a racer to the ArrayList racerList,
   *          increment yPos by the value of
   *          the constant local variable RACER_SPACE
   *
   *  if input is anything other than 'T', 't',
   *          'H' or 'h', do nothing
   */
   // write your switch statement here

   /** end of student code, part 1 */

} // end prepareToRace
```

Description

2. Next, write the code to display the racers at the starting position. Again, search for five asterisks (*****). This will position you inside the *getReady* method, which is called whenever a new *Racer* is added. The method draws the finish line and displays the racers at the starting position. The code to display the finish line is already written. For this task, you will write code to loop through the *ArrayList* of

Racers, calling the *draw* method for each racer. The portion of the *getReady* method where you will add your code is shown below.

```
/** getReady method
 *  @param gc   GraphicsContext context
 *  draws the finish line and draws the racers
 */
 protected void getReady( GraphicsContext gc )
 {
   // draw the finish line
   finishX = (int) canvas.getWidth( ) - 20;
   gc.setStroke( Color.BLUE );
   gc.strokeLine( finishX, 0, finishX, canvas.getHeight( ) );

   // display racers before race begins
   /* 2. ***** student writes this code
   *  loop through instance variable ArrayList racerList,
   *    which contains Racer object references,
   *    calling draw for each element. (Do not call move!)
   *  The API for draw is:
   *    void draw( GraphicsContext gc )
   *    where gc is the graphics context
   *    passed to this getReady method
   */
   // student code goes here

   /** end of student code, part 2 */

   } // end getReady
```

Description

3. Finally, write the code to run the race. Again, search for five asterisks (*****); this will position you inside the *runRace* method, which is called when the user presses the “Start Race” button. Task 3 is similar to task 2 in that you will loop through the *ArrayList* of *Racers*. In this task, however, you will call both the *move* and *draw* methods. The portion of the *runRace* method where you will add your code is shown below.

```
/** runRace method
 *  @param gc   GraphicsContext context
 *  moves and draws racers
 */
 public void runRace( GraphicsContext gc )
 {
 gc.setStroke( Color.BLUE );
 gc.strokeLine( finishX, 0, finishX, canvas.getHeight( ) );

 /* 3. ***** student writes this code
 *    loop through instance variable ArrayList racerList,
 *    which contains Racer object references,
 *    calling move, then draw for each element
 *  The API for move is:
 *      void move( )
 *  The API for draw is:
 *      void draw( GraphicsContext gc )
 *        where gc is the GraphicsContext object
 *        passed to this runRace method
 */
 // student code goes here

 /** end of student code, part 3 */

 } // end runRace
```

Description

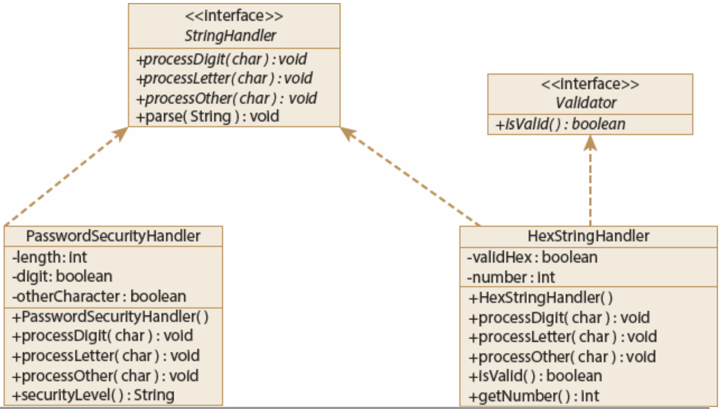
When you have finished writing the code, compile the source code and run the *PolymorphismApplication* file. Try several runs of the race with a different number of racers and with a different combination of *Tortoises* and *Hares*. **Figure 10.16** shows the race with four *Tortoises* and three *Hares*.

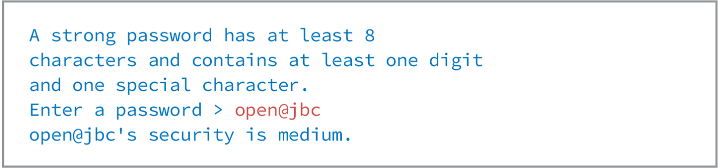
Figure 10.16 Another Run of the Race

Description

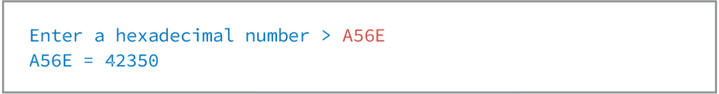
DISCUSSION QUESTIONS

1. Explain how polymorphism simplifies this application.
2. If you wanted to add another racer, for example, an aardvark, explain what code you would need to write and what existing code, if any, you would need to change.

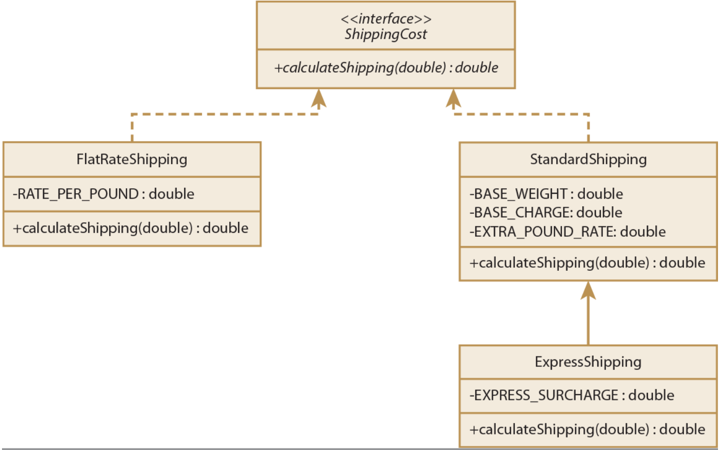

<<interface>>
StringHandler
+processDigit(char) : void
+processLetter(char) : void
+processOther(char) : void
+parse(String) : void
<<interface>>
Validator
+isValid( ) : boolean
PasswordSecurityHandler
-length: int
-digit: boolean
-otherCharacter : boolean
+PasswordSecurityHandler( )
+processDigit(char) : void
+processLetter(char) : void
+processOther(char) : void
+securityLevel( ) : String
HexStringHandler
-validHex : boolean
-number : int
+HexStringHandler( )
+processDigit(char) : void
+processLetter(char) : void
+processOther(char) : void
+isValid( ) : boolean
+getNumber( ) : int

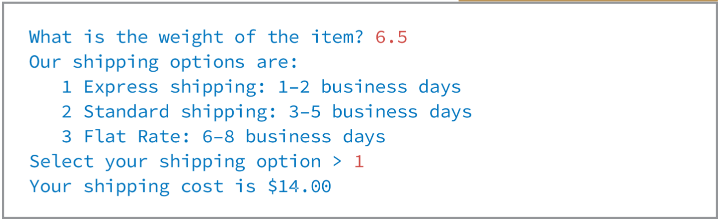


```
A strong password has at least 8
characters and contains at least one digit
and one special character.
Enter a password > open@jbc
open@jbc's security is medium.
```



```
Enter a hexadecimal number > A56E
A56E = 42350
```


<<interface>>
ShippingCost
+calculateShipping(double) : double
FlatRateShipping
-RATE_PER_POUND : double
+calculateShipping(double) : double
StandardShipping
-BASE_WEIGHT : double
-BASE_CHARGE: double
-EXTRA_POUND_RATE: double
+calculateShipping(double) : double
ExpressShipping
-EXPRESS_SURCHARGE : double
+calculateShipping(double) : double



```
What is the weight of the item? 6.5
Our shipping options are:
   1 Express shipping: 1-2 business days
   2 Standard shipping: 3-5 business days
   3 Flat Rate: 6-8 business days
Select your shipping option > 1
Your shipping cost is $14.00
```

10.8 Interfaces

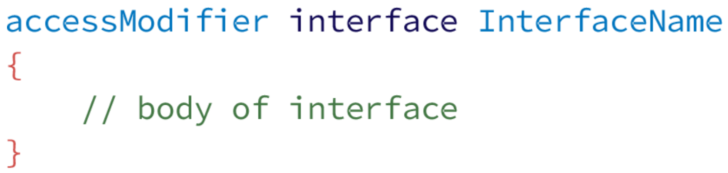
In Java, a class can inherit directly from only one class; that is, a class can *extend* only one class. To allow a class to inherit behavior from multiple sources, Java provides the **interface**. Interfaces are often used to provide a specification for performing common tasks. They can also be used to promote code reusability.

An interface typically specifies behavior that a class will *implement*.

Interface members can be any of the following:

- constants
- methods
- classes
- other interfaces

To define an interface, we use the following syntax:



```
accessModifier interface InterfaceName
{
    // body of interface
}
```

Description

Like classes, the Java convention is to name interfaces starting with a capital letter and capitalizing internal words.

All interfaces are *abstract*; thus, they cannot be instantiated. The *abstract* keyword, however, is usually omitted from the interface definition. If the interface's access modifier is *public*, its members are implicitly *public* as well.

SOFTWARE ENGINEERING TIP

When naming interfaces, start with a capital letter and capitalize any internal words.

Any field defined in a *public* interface is implicitly *public*, *static*, and *final*. These keywords can be specified, but typically we omit them because they are implicit. When we define a constant in the interface, we must also assign a value to that field. Note that because all fields are *static*, interfaces cannot have instance variables.

We can define various types of methods in interfaces:

- *public abstract* methods
- *public default* methods
- *public* or *private static* methods
- *private* methods

To define an *abstract* method, we provide only the method header followed by a semicolon. We do not provide a body for the method.

For the other method types, we do provide a method body; that is, *default, static,* and *private* methods are not *abstract*.

Private methods, introduced in Java version 9, allow us to define methods that provide a service to another method in the interface. For example, several methods in the interface may contain common code. By putting that common code into a *private* method, we can eliminate duplication of code. Each method can simply call the *private* method to access the needed functionality.

To inherit from an interface, our class declares that it *implements* the interface, using the following syntax:

```
accessModifier class ClassName extends SuperclassName
                                   implements Interface1, Interface2, ...
```

Description

The *extends* clause is optional if our class inherits only from the *Object* class. A class can *implement* 0, 1, or more interfaces. If our class *implements* more than one interface, the interfaces are specified in a comma-separated list of interface names.

When our class implements an interface, we inherit any *default* and *public static* methods, but we must provide method bodies for all the *abstract* methods.

Default methods, introduced in Java version 8, are *public* methods with a method body that all classes implementing the interface inherit. *Default* methods solve a problem: how can we add a new method to an existing interface? For example, assume we have defined an interface that specifies three *abstract* methods. Any class that implements our interface must provide method bodies for those three methods. Then suppose we want to add new functionality to the interface that would require adding a fourth method. If we add that fourth method as *abstract*, any existing class that implements the interface will no longer compile. We don’t want to break existing code!

So we have two options. The first option is to define a new interface that *extends* our original interface, that is, the new interface would inherit the original three *abstract* methods, and we would add the fourth method to the new interface as an *abstract* method. If existing classes want to take advantage of the new functionality, those classes would need to change their code to implement the new interface and thus add a body for the fourth method. The second simpler option is to add the fourth method to the original interface as a *default* method that provides some basic functionality. In this way, classes that already implement the original interface would automatically inherit the new functionality without needing to change their code. In addition, a class that implements an interface can optionally override any *default* methods. Thus, if the basic functionality is not appropriate, the implementing class can provide their own custom version of the method by overriding the *default* method.

Let’s consider the general problem of parsing a string of characters. We could have two applications: determining whether a password contains required characters, and converting a *String* containing hexadecimal digits to a decimal number. In each application, we need to parse (analyze) a *String* character by character. In the second application, we also need to validate the *String*.

Figure 10.17 shows the interfaces and classes used in these two applications. In this figure, we introduce the <<interface>> notation and a new UML symbol, the dotted line, which indicates a class that implements an interface.

Figure 10.17 UML for String Parsing Interfaces and Classes

Description

Each application needs to process the characters of the *String,* determining whether each character of the *String* is a digit, a letter, or another character (neither a digit nor a letter). But each application handles digits, letters, and other characters differently. We create an interface, *StringHandler,* shown in Example 10.21, that allows the implementing class to define a strategy for processing a *String*, one character at a time. The *StringHandler* interface specifies three *abstract* methods, and a *default parse* method that iterates through the *String* determining the type of each character, and calling the appropriate method. Any class that implements the *StringHandler* interface automatically inherits the *parse* method, and must implement the *processLetter*, *processDigit,* and *processOther* methods*.*

Example 10.21 The *StringHandler* Interface

```
 1 /*  StringHandler interface
 2 *   Anderson, Franceschi
 3 */
 4
 5 public interface StringHandler
 6 {
 7   // abstract methods to implement
 8   void processLetter( char c );
 9   void processDigit( char c );
10   void processOther( char c );
11
12   /** default parse method
13   * @param s the String to parse
14   */
15   default void parse( String s )
16   {
17     for ( int i = 0; i < s.length( ); i++ )
18     {
19       char c = s.charAt( i );
20       if ( Character.isDigit( c ) )
21         processDigit( c );
22       else if ( Character.isLetter( c ) )
23         processLetter( c );
24       else
25         processOther( c );
26     }
27   }
28 }
```

Description

The *parse* method, at lines 12–27, takes the *String* to parse as a parameter and walks through the *String*, a character at a time, determining the type of character, and calling the appropriate method of the *StringHandler* interface. In order to determine whether a character is a digit or a letter, the *parse* method calls two *static* methods of the *Character* wrapper class, *isDigit* and *isLetter*, shown in **Table 10.7**. Any class that implements the *StringHandler* interface must provide fully implemented versions of the three *abstract* methods:

processDigit, *processLetter*, and *processOther*. Additionally, if the implementing class needs to process the *String* in a different order, perhaps from the last letter to the first, that class can write its own version of the *parse* method that overrides the *parse* method in the interface.

Some *Static* Methods of the *Character* Wrapper Class

Useful *Static* Methods of the *Character* Class

Return value	Method name and argument list
`boolean`	`isDigit( char ch )` returns *true* if *ch* represents a digit from ’0’ to ‘9’; *false* otherwise.
`boolean`	`isLetter( char ch )` returns *true* if *ch* represents a letter from ‘a’ to ‘z’ or ‘A’ to ‘Z’; *false* otherwise.
`int`	`getNumericValue( char ch )` returns the integer value of the specified Unicode character ch.

Description

We are now ready to design a specialized class that implements the *StringHandler* interface to parse the string with a particular objective in mind. In our example, the first application processes a password to determine its strength. A strong password must contain at least eight characters, at least one digit, and at least one special character. A password containing fewer than six characters is considered weak; otherwise, we consider that the password security level is medium.

The *PasswordSecurityHandler* class, shown in **Example 10.22**, implements the *StringHandler* interface (line 9); thus, it provides implementations of the *processLetter, processDigit*, and *processOther* methods. The *default parse* method meets this application's needs, so we do not override the *default* implementation of the *parse* method.

Example 10.22 The *PasswordSecurityHandler* Class

```
/*  PasswordSecurityHandler class
 *  Implements methods of the StringHandler interface
 *  to parse a String containing a password.
 *
 *  Anderson, Franceschi
*/

public class PasswordSecurityHandler
                    implements StringHandler
{
  private int length;
  private boolean digit;
  private boolean otherCharacter;

  /** default constructor
  *  sets length to 0, digit and otherCharacter flags to false
  */
  public PasswordSecurityHandler( )
  {
    length = 0;
    digit = false;
    otherCharacter = false;
  }

  /** processLetter method
  * @param c  character to process
  * adds 1 to length
  */
  @Override
  public void processLetter( char c )
  {
    length++;
  }

  /** processDigit method
  * @param c  character to process
  * adds 1 to length, sets digit flag to true
  */
  @Override
  public void processDigit( char c )
  {
    length++;
    digit = true;
  }

  /** processOther method
  * @param c  character to process
  * adds 1 to length,
  * sets otherCharacter flag to true
  */
  @Override
  public void processOther( char c )
  {
    length++;
    otherCharacter = true;
  }

  /** securityLevel method
  * @return  "weak" if password contains fewer than 6 characters
  *          "strong" if password has at least 8 characters, at least
  *                  one digit, and at least one other character
  *                  that is neither a letter nor a digit
  *          "medium" otherwise
  */
  public String securityLevel( )
  {
    if ( length < 6 )
      return "weak";
    else if ( length >= 8 && digit && otherCharacter )
      return "strong";
    else
      return "medium";
  }
}
```

Description

We define three instance variables (lines 11–13): *length* to hold the number of characters in the password and two flag variables, *digit* and *otherCharacter,* which will indicate whether a digit and a special character were found in the password. The constructor (lines 15–23) initializes *length* to 0 and the flag variables to *false*. The *processLetter* method (lines 25–33) simply increments the *length* variable. The *processDigit* method (lines 35–44) increments *length* and also sets the *digit* flag to *true* to indicate that at least one digit has been found. Similarly, the *processOther* method (lines 46–56) increments *length* and sets the *otherCharacter* flag to *true* to indicate that at least one special character has been found.

The *securityLevel* method (lines 58–73) tests the strength of the password based on its length and the *digit* and *otherCharacter* flags that were set as the password was processed and returns either “weak,” “medium,” or “strong.”

Example 10.23 shows a client program for parsing a password. We ask the user to enter a password at lines 12–16. At line 18, we instantiate a *PasswordSecurityHandler* object. We then call the *parse* method (line 19), passing the password entered by the user. After the password has been parsed, we call the *securityLevel* method of the *PasswordSecurityHandler* class and output the strength of the password (lines 21–22). **Figure 10.18** shows the output of Example 10.23 when the user enters the password *open@jbc*.

Figure 10.18 Output from Example 10.23

Description

Example 10.23 The *PasswordSecurityHandlerClient* Class

```
/* PasswordSecurityHandlerClient
* Anderson, Franceschi
*/

import java.util.Scanner;

public class PasswordSecurityHandlerClient
{
  public static void main( String [ ] args )
  {
    Scanner scan = new Scanner( System.in );
    System.out.println( "A strong password has at least 8 \n"
          + "characters and contains at least one digit \n"
          + "and one special character." );
    System.out.print( "Enter a password > " );
    String password = scan.next( );

    PasswordSecurityHandler psh = new PasswordSecurityHandler( );
    psh.parse( password );

    System.out.println( password + "'s security is "
               + psh.securityLevel( ) );
  }
}
```

Description

For the second application, where we analyze a *String* representing a hexadecimal number, we reuse the *StringHandler* interface. This application determines whether the *String* contains only valid hexadecimal characters and calculates the decimal equivalent of the hex value represented by the *String*.

As we discussed earlier, a class can implement several interfaces. We now define another interface, *Validator*, shown in Example 10.24, which specifies one *abstract* method to implement: *isValid*. We created the *Validator* interface because determining whether a string is valid is a generic operation and not necessarily specific to parsing a hexadecimal number. Thus, other classes can also implement the *Validator* interface.

Example 10.24 The *Validator* Interface

```
 1 /** Validator interface
 2 *   Anderson, Franceschi
 3 */
 4 
 5 @FunctionalInterface
 6 public interface Validator
 7 {
 8   // abstract method to implement
 9   boolean isValid( );
10 }
```

Description

If the interface has only one *abstract* method to implement, the interface is called a **functional interface** and can be used with **lambda expressions.** We cover lambda expressions later in the text. To ensure that the interface we are defining is, in fact, **a functional interface,** we can precede the definition with the *@FunctionalInterface* annotation. Because *Validator* has only one method to implement and is thus a functional interface, we precede its definition with the above annotation (line 5).

The *HexStringHandler* class, shown in **Example 10.25**, implements both the *StringHandler* and the *Validator* interfaces (line 11). Again, the *default parse* method meets our needs, so we do not override that method. Our *HexStringHandler* class has two instance variables (lines 13–14): a flag variable, *validHex*, which will indicate whether the string being parsed contains only valid hex characters, and *number*, which will hold the decimal equivalent of the hex value. The default constructor (lines 16–24) initializes these variables to *true* and *0*, respectively. The decimal equivalent of the hex number is calculated as each character of the *String* is parsed in the *processLetter* method (providing that the letter is a valid hex digit) and the *processDigit* method. The *processLetter* and *processDigit* methods use the *static getNumericValue* method of the *Character* wrapper class, also shown in Table 10.7, to retrieve the numeric equivalent of the character being processed. If the *processLetter* method (lines 26–40) finds that the letter is not valid or if the *processOther* method (lines 53–62) is called—indicating that the character is neither a letter nor a digit—the *validHex* flag is set to *false*.

Example 10.25 *HexStringHandler* Class

```
/* HexStringHandler class
*  Implements the StringHandler interface
*  to parse a String that contains a hex number
*  into its decimal equivalent
*  Implements the Validator interface to determine
*  validity of String
*  Anderson, Franceschi
*/

public class HexStringHandler
               implements StringHandler, Validator
{
  private boolean validHex;
  private int number;

  /** default constructor
  *  initializes number to 0
  *  and validHex to true
  */
  public HexStringHandler( )
  {
    validHex = true;
    number = 0;
  }

  /** processLetter method
  * @param c  the character to process
  * if c is between 10 and 15 (hex A through F),
  * uses its value to update the decimal value (number)
  * otherwise, character is invalid letter
  */
  @Override
  public void processLetter( char c )
  {
    int n = Character.getNumericValue( c );
    if ( n >= 10 && n <= 15 ) // valid hex character?
      number = 16 * number + n; // update number
    else // invalid hex character
      validHex = false;
  }

  /** processDigit method
  * @param c  the character to process
  * uses numeric value of c to update the decimal value (number)
  */
  @Override
  public void processDigit( char c )
  {
    int n = Character.getNumericValue( c );
    number = 16 * number + n; // update number
  }

  /** processOther method
  * @param c  the character to process
  * character is not a valid hex digit
  */
  @Override
  public void processOther( char c )
  {
    // c is an invalid hex character
    validHex = false;
  }

  /** isValid method
  * @return true if all characters
  * in String are valid hex characters,
  * else returns false
  */
  public boolean isValid( )
  {
    return validHex;
  }

  /** getNumber method
  * @return if valid, returns the calculated decimal value
  * else, returns -1
  */
  public int getNumber( )
  {
    if ( isValid( ) )
      return number;
    else
      return -1;
  }
}
```

Description

At lines 64–72, the *isValid* method returns *true* if the parsed string represents a valid hexadecimal number, and *false* otherwise. At lines 74–84, the *getNumber* method returns the decimal number if the *String* parsed contains only hexadecimal digits, and *–1* otherwise.

Example 10.26 shows a similar client program for the hex conversion application. We ask the user to enter a hexadecimal number at lines 12–13. At lines 15–16, we instantiate a *HexStringHandler.* At line 16, we call the *parse* method, passing it the hex *String* entered by the user. After parsing the *String*, we test if the *String* entered by the user is a valid hexadecimal number by calling the *isValid* method of the *HexStringHandler* class (line 18). If it is valid, we call the *getNumber* method of the *HexStringHandler* class to output the decimal equivalent of that number (line 19); otherwise, we output a message that the hex number was not valid (line 21). **Figure 10.19** shows the output of Example 10.26 when the user enters the valid hex number *A56E*.

Figure 10.19 Output from Example 10.26

Description

Example 10.26 The *HexStringHandlerClient* Class

```
/*  HexStringHandlerClient class
*   Anderson, Franceschi
*/

import java.util.Scanner;

public class HexStringHandlerClient
 {
  public static void main( String [ ] args )
  {
    Scanner scan = new Scanner( System.in );
    System.out.print( "Enter a hexadecimal number > " );
    String hex = scan.next( );

    HexStringHandler hsh = new HexStringHandler( );
    hsh.parse( hex );

    if ( hsh.isValid( ) )
      System.out.println( hex + " = " + hsh.getNumber( ) );
    else
      System.out.println( hex + " is not a valid hex number." );
  }
}
```

Description

Polymorphism can also be used with interfaces. Although interfaces cannot be instantiated, we can assign an object of a class that implements an interface to an interface reference, and we use that reference to call the methods of the interface.

To illustrate polymorphism with interfaces, let's assume that we are a company that needs to calculate shipping costs for items our customers purchase. We have decided to offer three different pricing strategies based on the weight of the item. With flat-rate shipping, we charge a fixed fee per pound. For standard shipping, we charge a base rate for the first 5 pounds, then a fixed rate for each additional pound. And for express shipping, we add a fixed fee to the standard shipping charge.

Note that in each case, we are calculating a price based on weight. To handle this situation, we can create a *ShippingCost* interface and derive three classes that implement the *ShippingCost* interface, each performing its appropriate shipping cost calculation. The UML for this application is shown in **Figure 10.20**.

Figure 10.20 UML for the Shipping Strategy

Description

The *ShippingCost* interface is shown in **Example 10.27**. This interface has one *abstract* method, *calculateShipping*, that accepts the weight as a parameter and returns the shipping cost (line 9).

Example 10.27 The *ShippingCost* Interface

```
1 /** Shipping Cost
2 *   Anderson, Franceschi
3 */
4
5 @FunctionalInterface
6 public interface ShippingCost
7 {
8    double calculateShipping( double weight );
9 }
```

Description

We can now define three classes that implement the *ShippingCost* interface: *FlatRateShipping*, *StandardShipping*, and *ExpressShipping*, with each providing a custom version of the *calculateShipping* method.

Example 10.28 shows the *FlatRateShipping* class. In line 6, we specify that this class implements the *ShippingCost* interface, and in lines 10–14 we provide the *calculateShipping* method, which simply returns the weight multiplied by a constant rate.

Example 10.28 The *FlatRateShipping* Class

```
/** FlatRateShipping
*    cost is .50/pound
*    Anderson, Franceschi
*/

public class FlatRateShipping implements ShippingCost
{
   private final double RATE_PER_POUND = .50;

   @Override
   public double calculateShipping( double weight )
   {
      return RATE_PER_POUND * weight;
   }
}
```

Description

Example 10.29 shows the *StandardShipping* class, which also implements the *ShippingCost* interface (line 7). Its *calculateShipping* method (lines 13–21) computes a fixed charge for the first 5 pounds, plus a dollar for each additional pound.

Example 10.29 The *StandardShipping* Class

```
 1 /** StandardShipping
 2 *    cost is $2.50 for the first 5 pounds,
 3 *       and $1 for each additonal pound
 4 *    Anderson, Franceschi
 5 */
 6
 7 public class StandardShipping implements ShippingCost
 8 {
 9     private final double BASE_WEIGHT = 5;
10     private final double BASE_CHARGE = 2.5;
11     private final double EXTRA_POUND_RATE = 1.00;
12
13    @Override
14    public double calculateShipping( double weight )
15    {
16       if ( weight > BASE_WEIGHT )
17         return BASE_CHARGE
18              +  ( ( weight - BASE_WEIGHT ) * EXTRA_POUND_RATE );
19       else
20         return BASE_CHARGE;
21    }
22 }
```

Description

Because for express shipping we charge the standard rate plus a fixed fee, we want to use the same calculation as the standard shipping and just add the fixed fee. Rather than repeat the code for the standard shipping calculation, we can define the *ExpressShipping* class as extending the *StandardShipping* class. **Example 10.30** shows the *ExpressShipping* class. In line 6, we specify that *ExpressShipping* inherits from *StandardShipping*. Note that we don’t need to specify that this class implements the *ShippingCost* interface because *StandardShipping* already does that. Any subclass automatically implements any interfaces implemented by the superclass. In lines 10–14, we override the *calculateShipping* method defined in the *StandardShipping* superclass. Inside our method, we call the version of the *calculateShipping* method in the *StandardShipping* class to determine the standard fee (line 13) and add the express fixed fee to the returned value.

Example 10.30 The *ExpressShipping* Class

```
 1 /** ExpressShipping
 2 *   cost is $10, in addition to standard shipping
 3 *   Anderson, Franceschi
 4 */
 5
 6 public class ExpressShipping extends StandardShipping
 7 {
 8    private final double EXPRESS_SURCHARGE = 10;
 9
10    @Override
11    public double calculateShipping( double weight )
12    {
13       return EXPRESS_SURCHARGE + super.calculateShipping( weight );
14    }
15 }
```

Description

Putting all this together, **Example 10.31** shows a client class, *ShippingCalculator*, that allows a user to select a shipping option and then calculates the shipping cost for an item. At lines 17–22, we prompt the user for the weight of the item and provide a menu from which the user can select a shipping option. Then, using a *switch* expression (lines 24–30), we declare a reference to the *ShippingCost* interface and assign an object reference of the appropriate class to it (lines 26, 27, and 28). If the user selects a valid option, we call the *calculateShipping* method using the *shippingCost* reference (lines 33–34). As a result, the correct method for the selected pricing strategy is called at runtime. **Figure 10.21** shows the output of the program when the user selects express shipping for an item that weighs 6.5 pounds.

Figure 10.21 Output from Example 10.31

Description

This application meets the four requirements for polymorphism. All classes are in the same hierarchy; the subclasses override the same method; we assign subclass references to a superclass/interface reference; and we use the superclass/interface reference to call the method.

Example 10.31 The *ShippingCalculator* Class

```
 1 /** ShippingCalculator
 2 *   Anderson, Franceschi
 3 */
 4 import java.util.Scanner;
 5 import java.text.NumberFormat;
 6
 7 public class ShippingCalculator
 8 {
 9    public static void main( String [ ] args )
10    {
11      Scanner scan = new Scanner( System.in );
12      NumberFormat money = NumberFormat.getCurrencyInstance( );
13
14      System.out.print( "What is the weight of the item? " );
15      double weight = scan.nextDouble( );
16
17      System.out.println( "Our shipping options are: "
18                          + "\n\t1 Express shipping: 1-2 business days"
19                          + "\n\t2 Standard shipping: 3-5 business days"
20                          + "\n\t3 Flat Rate: 6-8 business days" );
21      System.out.print( "Select your shipping option > " );
22      int option = scan.nextInt( );
23
24      ShippingCost shippingCost = switch ( option )
25      {
26         case 1 -> new ExpressShipping( );
27         case 2 -> new StandardShipping( );
28         case 3 -> new FlatRateShipping( );
29         default -> null;
30      };
31
32      if ( shippingCost != null )
33         System.out.println( "Your shipping cost is "
34             + money.format( shippingCost.calculateShipping( weight ) ) );
35      else
36         System.out.println( "No valid shipping option selected." );
37    }
38 }
```

Description

with these end-of-chapter questions

10.10.1	Multiple Choice Exercises Questions 3, 5, 6
10.10.3	Fill In the Code Question 25
10.10.4	Identifying Errors in Code Questions 26, 27, 28, 29, 30
10.10.8	Technical Writing Questions 56, 57

CODE IN ACTION

Within the online resources, you will find a movie with a step-by-step illustration of the use of *abstract* classes and interfaces in a program. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

© Lois GoBe/Shutterstock.

CHAPTER REVIEW

10.9 Chapter Summary

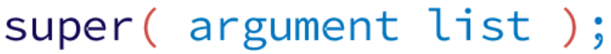
- Inheritance lets us organize related classes into ordered levels of functionality, called hierarchies. The advantage is that we write the common code only once and reuse it in multiple classes.
- A subclass inherits methods and fields of its superclass. A subclass can have only one direct superclass, but many subclasses can inherit from a common superclass.
- Inheritance implements the “is a” relationship between classes. Any object of a subclass is also an object of the superclass.
- All classes inherit from the *Object* class.
- To specify that a subclass inherits from a superclass, the subclass uses the *extends* keyword in the class definition, as in the following syntax:

```
accessModifier class ClassName extends SuperclassName
```

Description

- A subclass does not inherit constructors or *private* members of the superclass. However, the superclass constructors are still available to be called from the subclass, and the *private* fields of the superclass are implemented as fields of the subclass object.

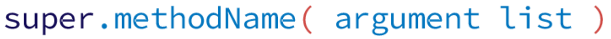
- To access *private* fields of the superclass, the subclass needs to call the accessor and mutator methods provided by the superclass.
- To call the constructor of the superclass, the subclass constructor uses the following syntax:

  ```
  super( argument list );
  ```

  Description

  If used, this statement must be the first statement in the subclass constructor.
- A subclass can override an inherited method by providing a new version of the method. The new method's API must be identical to the inherited method. To call the inherited version of the method, the subclass uses the *super* object reference using the following syntax:

  ```
  super.methodName( argument list )
  ```

  Description
- Any field declared using the *protected* access modifier is inherited by the subclass. As such, the subclass can directly access the field without calling its accessor or mutator method.
- An *abstract* class can be used to specify APIs for methods that subclasses should implement. An *abstract* class cannot be used to instantiate objects. A class is declared to be *abstract* by including the *abstract* keyword in the class header.
- An *abstract* class typically has one or more *abstract* methods. An *abstract* method specifies the API of the method, but does not provide an implementation. The API of an *abstract* method is followed by a semicolon.
- When a subclass inherits from an *abstract* class, it can provide implementations for any, all, or none of the *abstract* methods. If the subclass does not implement all the *abstract* methods of the superclass, then the subclass must

also be declared as *abstract*. If, however, the subclass implements all the *abstract* methods in the superclass and is not declared *abstract*, then the class is not *abstract* and we can instantiate objects of that subclass.

- Polymorphism simplifies the processing of various objects in a hierarchy by allowing us to use the same method call for any object in the hierarchy. We assign an object reference of a subclass to a superclass reference, then make the method call using the superclass object reference. At run time, the JVM determines to which class in the hierarchy the object actually belongs and calls the appropriate version of the method for that class.
- Interfaces allow a class to inherit behavior from multiple sources. Interface members can be classes, constants, *abstract* methods, *default* methods, *private* or *public* non-*abstract* *static* methods, *private* non-*static* non-*abstract* methods, or other interfaces.
- To define an interface, use the following syntax:

```
accessModifier interface InterfaceName
{
  // body of interface
}
```

Description

- To use an interface, a class header includes the *implements* keyword and the name of the interface, as in the following syntax:

```
accessModifier class ClassName implements InterfaceName
```

Description

- A class that implements an interface must provide full implementations of any *abstract* methods, and can either accept or override any *default* methods.
- To specify that a subclass both inherits from a superclass and uses an interface, a class header includes both the *extends* and the *implements* keywords as in the syntax that follows:

```
accessModifier class ClassName extends SuperclassName
                               implements Interface1, Interface2, ...
```

Description

10.10 Exercises, Problems, and Projects

10.10.1 Multiple Choice Exercises

1. The *extends* keyword applies to
    - a class inheriting from another class.
    - a variable.
    - a method.
    - an expression.

6. A Java class can inherit from two or more classes.
    - true
    - false

9. In Java, multiple inheritance is implemented using the concept of
    - an interface.
    - an *abstract* class.
    - a *private* class.

13. Which of the following is inherited by a subclass?
    - all instance variables and methods
    - *public* instance variables and methods only
    - *protected* instance variables and methods only
    - *protected* and *public* instance variables and methods

18. What Java keyword is used in a class header when a class is defined as inheriting from an interface?
    - *inherits*
    - *includes*

- *extends*
- *implements*

23. A Java class can implement one or more interfaces.

    - true
    - false

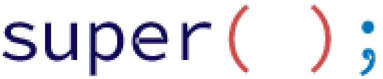
26. How do you instantiate an object from an *abstract* class?

    - With any constructor.
    - With the default constructor only.
    - You cannot instantiate an object from an *abstract* class.

30. When a class overrides a method, what object reference is used to call the method inherited from the superclass?

    - *inherited*
    - *super*
    - *class*
    - *methodName*

35. Where should the following statement be located in the body of a subclass constructor?



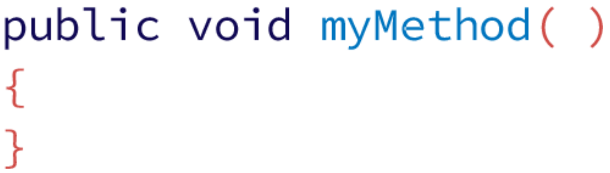
```
super( );
```

    - It should be the last statement.
    - It should be the first statement.
    - It can be anywhere.

39. If a class contains an *abstract* method, then

    - the class must be declared *abstract*.
    - the class is not *abstract*.
    - the class may or may not be *abstract*.
    - all of the above

44. What can you tell about the following method?



```
public void myMethod( )
{
}
```

Description

- This method is *abstract*.
- This method is not *abstract*.

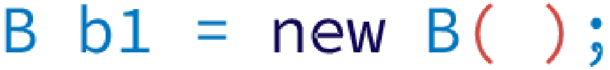
10.10.2 Reading and Understanding Code

For Questions 12 to 20, consider the following three classes:

```
public class A

{
  private int number;
  protected String name;
  public double price;

  public A( )
  {
   System.out.println( "A( ) called" );
  }

  private void foo1( )
  {
   System.out.println( "A version of foo1( ) called" );
  }

  protected int foo2( )
  {
   System.out.println( "A version of foo2( ) called" );
   return number;
  }

  public String foo3( )
  {
   System.out.println( "A version of foo3( ) called" );
   return "Hi";
  }
}
public class B extends A
{
 private char service;
 public B( )
 {
   super( );
   System.out.println( "B( ) called" );
 }

 public void foo1( )
 {
   System.out.println( "B version of foo1( ) called" );
 }

 protected int foo2( )
 {
   int n = super.foo2( );
   System.out.println( "B version of foo2( ) called" );
   return ( n + 5 );
 }
 public String foo3( )
 {
   String temp = super.foo3( );
   System.out.println( "B version of foo3( )" );
   return ( temp + " foo3" );
  }
}

public class C extends B
{
  public C( )
  {
   super( );
   System.out.println( "C( ) called" );
  }
  public void foo1( )
  {
   System.out.println( "C version of foo1( ) called" );
  }
}
```

Description

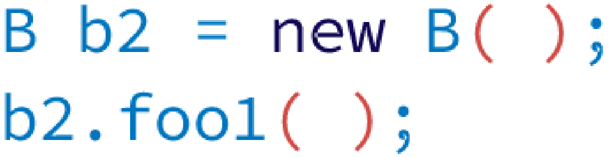
1. Draw the UML diagram for the class hierarchy.
2. What fields and methods are inherited by which class?
3. What fields and methods are not inherited?
4. What is the output of the following code sequence?



```
B b1 = new B( );
```

Description

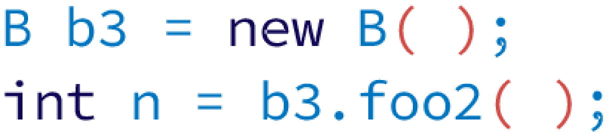
5. What is the output of the following code sequence?



```
B b2 = new B( );
b2.foo1( );
```

Description

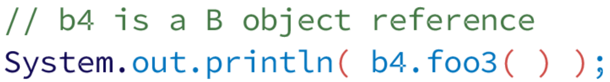
6. What is the output of the following code sequence?



```
B b3 = new B( );
int n = b3.foo2( );
```

Description

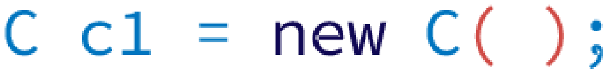
7. What is the output of the following code sequence?



```
// b4 is a B object reference
System.out.println( b4.foo3( ) );
```

Description

8. What is the output of the following code sequence?



```
C c1 = new C( );
```

Description

9. What is the output of the following code sequence?

```
// c2 is a C object reference
c2.foo1( );
```

Description

10.10.3 Fill In the Code

For Questions 21 to 25, consider the following class *F* and the interface *I*:

```
public class F
{
 private String first;
 protected String name;

 public F( )
 { }

public F( String first, String name )
{
  this.first = first;
  this.name = name;
}

public String getFirst( )
{
 return first;
}
public String getName( )
{
 return name;
}
public String toString( )
{
 return  "first: " + first + "\tname: " + name ;
}
public boolean equals( Object f )
{
 if ( ! ( f instanceof F ) )
     return false;
 else
 {
     F objF = ( F ) f;
     return( first.equals( objF.first ) && name.equals( objF.name ) );
  }
 }
}
public interface I
{
  String TYPE = "human";
 int age( );
}
```

Description

1. The *G* class inherits from the *F* class. Code the class header of the *G* class.

```
// your code goes here
```

2. Inside the *G* class, which inherits from the *F* class, declare a *private* instance variable for the middle initial and code a constructor with three parameters, calling the constructor of the *F* class and assigning the third parameter, a *char*, to the new instance variable.

```
// your code goes here
```

3. Inside the *G* class, which inherits from the *F* class, code the *toString* method, which returns a printable representation of a *G* object reference.

```
// your code goes here
```

4. Inside the *G* class, which inherits from the *F* class, code the *equals* method, which compares two *G* objects and returns *true* if they have identical instance variables; *false* otherwise.

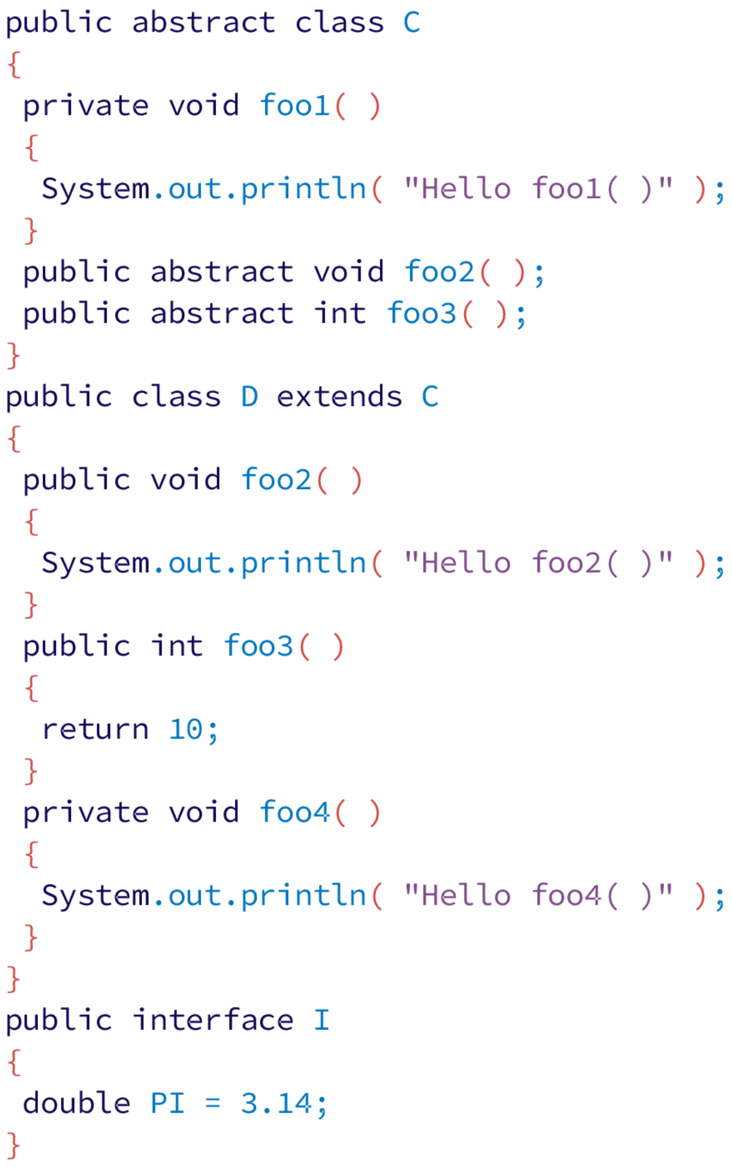
```
// your code goes here
```

5. The *K* class inherits from the *F* class and the *I* interface; code the class header of the *K* class.

```
// your code goes here
```

10.10.4 Identifying Errors in Code

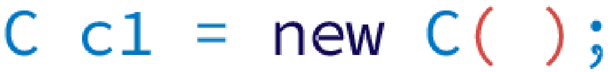
For Questions 26 to 31, consider the following two classes, *C* and *D*, and interface *I*:



```
public abstract class C
{
 private void foo1( )
 {
  System.out.println( "Hello foo1( )" );
 }
 public abstract void foo2( );
 public abstract int foo3( );
}
public class D extends C
{
 public void foo2( )
 {
  System.out.println( "Hello foo2( )" );
 }
 public int foo3( )
 {
  return 10;
 }
 private void foo4( )
 {
  System.out.println( "Hello foo4( )" );
 }
}
public interface I
{
 double PI = 3.14;
}
```

Description

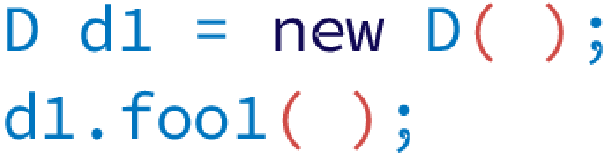
1. Where is the error in this code sequence?



```
C c1 = new C( );
```

Description

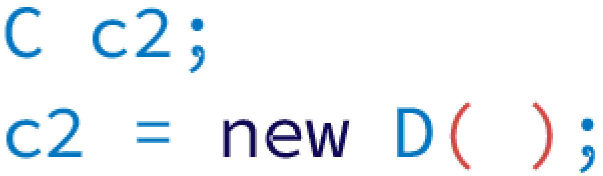
2. Where is the error in this code sequence?



```
D d1 = new D( );
d1.foo1( );
```

Description

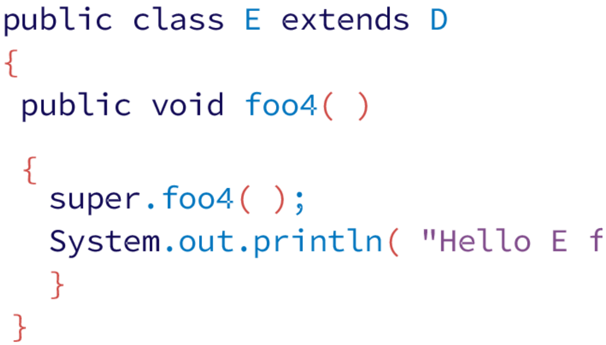
3. Is there an error in this code sequence? Why or why not?



```
C c2;
c2 = new D( );
```

Description

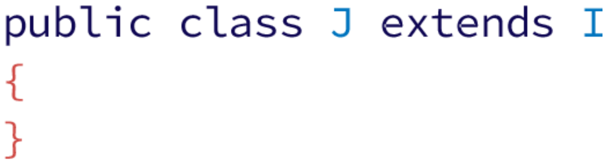
4. Where is the error in this new class?



```
public class E extends D
{
 public void foo4( )
 {
   super.foo4( );
   System.out.println( "Hello E f
   }
 }
```

Description

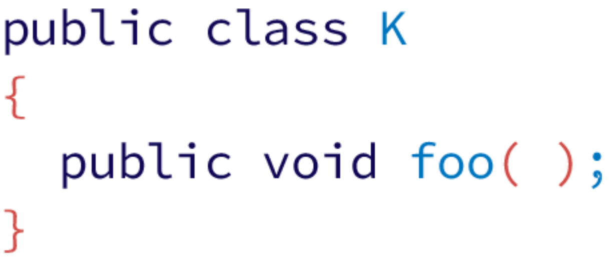
5. Where is the error in this class?



```
public class J extends I
{
}
```

Description

6. Where is the error in this class?



```
public class K
{
   public void foo( );
}
```

Description

10.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

For questions 32 to 35, explain what the problem is and how to fix it.

1. You coded the following class:

```
public class Z extends String, Integer
{
}
```

Description

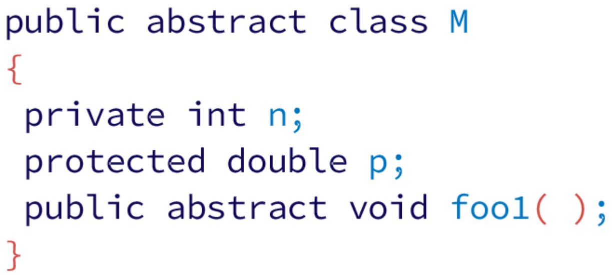
When you compile, you get the following message:

```
Z.java:1: error: '{' expected
public class Z extends String, Integer
                             ^
1 error
```

Description

Explain what the problem is and how to fix it.

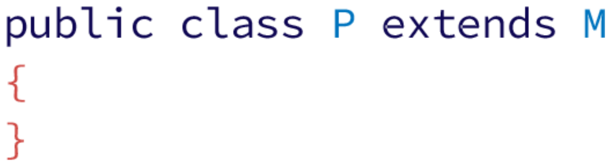
For Exercises 33 to 35, consider the following class:



```
public abstract class M
{
 private int n;
 protected double p;
 public abstract void foo1( );
}
```

Description

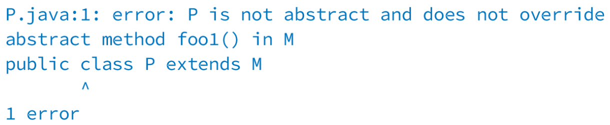
2. You coded the following class:



```
public class P extends M
{
}
```

Description

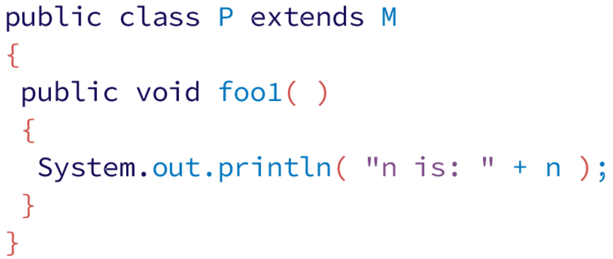
When you compile, you get the following message:



```
P.java:1: error: P is not abstract and does not override
abstract method foo1() in M
public class P extends M
       ^
1 error
```

Description

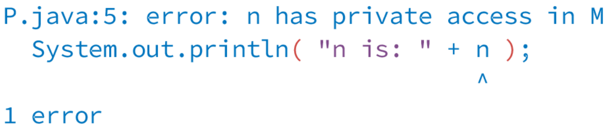
3. You coded the following class:



```
public class P extends M
{
 public void foo1( )
 {
  System.out.println( "n is: " + n );
 }
}
```

Description

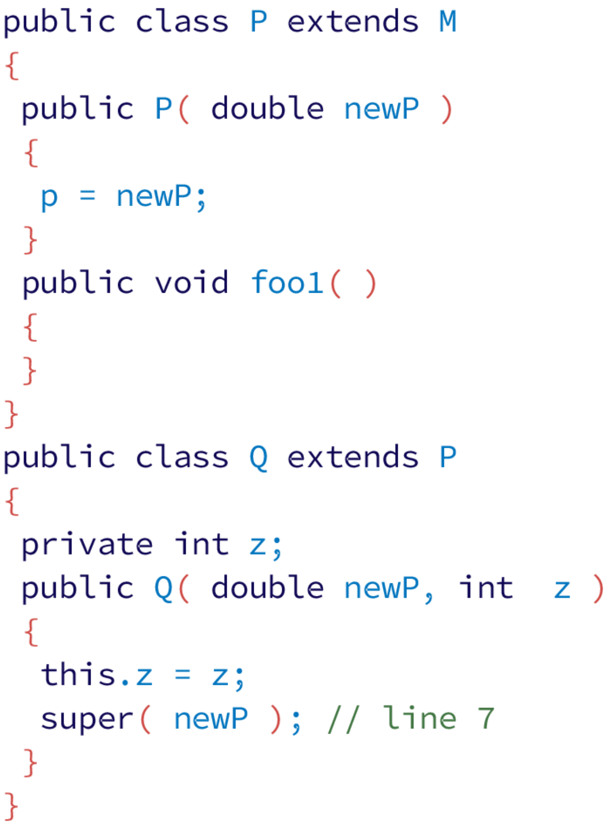
When you compile, you get the following message:



```
P.java:5: error: n has private access in M
  System.out.println( "n is: " + n );
                                 ^
1 error
```

Description

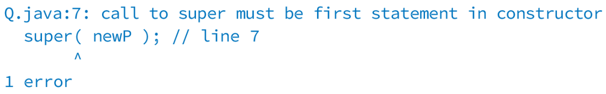
4. You coded the following classes:



```
public class P extends M
{
 public P( double newP )
 {
  p = newP;
 }
 public void foo1( )
 {
 }
}
public class Q extends P
{
 private int z;
 public Q( double newP, int  z )
 {
  this.z = z;
  super( newP ); // line 7
 }
}
```

Description

When you compile, you get the following message:



```
Q.java:7: call to super must be first statement in constructor
  super( newP ); // line 7
       ^
1 error
```

Description

10.10.6 Write a Short Program

For Exercises 36 to 40, consider the following class:

```
   public class Game
   {
    private String description;
    public Game( String description )
    {
     setDescription( description );
    }
    public String getDescription( )
    {
     return description;
    }
   public void setDescription( String description )
   {
    this.description = description;
   }
    public String toString( )
   {
    return "description: " + description ;
   }
  }
```

Description

1. Write a class encapsulating a PC-based game, which inherits from *Game*. A PC-based game has the following additional attributes: the minimum megabytes of RAM needed to play the game, the number of megabytes needed on the hard drive to install the game, and the minimum GHz performance of the CPU. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.
2. Write a class encapsulating a board game, which inherits from *Game*. A board game has the following additional attributes: the number of players and whether the game can end in a tie. Code the constructor and the *toString*

method of the new class. You also need to include a client class to test your code.

3. Write a class encapsulating a sports game, which inherits from *Game*. A sports game has the following additional attributes: whether the game is a team or individual game, and whether the game can end in a tie. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.
4. Write a class encapsulating a trivia game, which inherits from *Game*. A trivia game has the following additional attributes: the ultimate money prize and the number of questions that must be answered to win the ultimate money. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.
5. Write a class encapsulating a board game, which inherits from *Game*. A board game has the following additional attributes: the minimum number of players, the maximum number of players, and whether there is a time limit to finish the game. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.

   For Exercises 41 to 45, consider the following class:

```
public class Store
{
 public final double SALES_TAX_RATE = 0.06;
 private String name;
 public Store( String name )
 {
  setName( name );
 }
 public String getName( )
 {
  return name;
 }
 public void setName( String name )
 {

  this.name = name;
 }
 public String toString( )
 {
  return ( "name: " + name );
 }
}
```

Description

6. Write a class encapsulating a web store, which inherits from *Store*. A web store has the following additional attributes: an Internet address and the programming language in which the website was written. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.
7. Write a class encapsulating a music store, which inherits from *Store*. A music store has the following additional attributes: the number of titles it offers and its address. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.
8. Write a class encapsulating a bike store, which inherits from *Store*. A bike store has the following additional

attributes: the number of bicycle brands that it carries and whether it sponsors a bike club. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.

9. Write a class encapsulating a grocery store, which inherits from *Store*. A grocery store has the following additional attributes: annual revenues and whether it is an independent store or part of a chain. Code the constructor and the *toString* method of the new class; also code a method returning the annual taxes paid by the store. You also need to include a client class to test your code.
10. Write a class encapsulating a restaurant, which inherits from *Store*. A restaurant has the following additional attributes: how many people are served every year and the average price per person. Code the constructor and the *toString* method of the new class; also code a method returning the average taxes per year. You also need to include a client class to test your code.

10.10.7 Programming Projects

1. Write a superclass encapsulating a rectangle. A rectangle has two attributes representing the width and the height of the rectangle. It has methods returning the perimeter and the area of the rectangle. This class has a subclass, encapsulating a parallelepiped, or box. A parallelepiped has a rectangle as its base, and another attribute, its length; it has two methods that calculate and return its area and volume. You also need to include a client class to test these two classes.
2. Write a superclass encapsulating a circle; this class has one attribute representing the radius of the circle. It has

methods returning the perimeter and the area of the circle. This class has a subclass, encapsulating a cylinder. A cylinder has a circle as its base, and another attribute, its length; it has two methods, calculating and returning its area and volume. You also need to include a client class to test these two classes.

3. Write an *abstract* superclass encapsulating a shape: A shape has two *abstract* methods: one returning the perimeter of the shape, another returning the area of the shape. It also has a constant field named PI. This class has two non-*abstract* subclasses: one encapsulating a circle, and the other encapsulating a rectangle. A circle has one additional attribute, its radius. A rectangle has two additional attributes, its width and height. You also need to include a client class to test these two classes.
4. Write an *abstract* superclass encapsulating a vehicle: A vehicle has two attributes: its owner's name and its number of wheels. This class has two *non-abstract* subclasses: one encapsulating a bicycle, and the other encapsulating a motorized vehicle. A motorized vehicle has the following additional attributes: its engine volume displacement, in liters; and a method computing and returning a measure of horsepower—the number of liters times the number of wheels. You also need to include a client class to test these two classes.
5. Write an *abstract* superclass encapsulating some food; it has two attributes: its description and the number of calories per serving. It also has an *abstract* method taking a number of servings as a parameter and returning the number of calories. This class has two non-*abstract* subclasses: one encapsulating a liquid food (such as a

drink, for instance), and the other encapsulating a fruit. A liquid food has an additional attribute: its viscosity. A fruit has an additional attribute: its season. You also need to include a client class to test these two classes.

6. Write an *abstract* superclass encapsulating a college applicant: A college applicant has two attributes: the applicant's name and the college the applicant is applying to. This class has two non-*abstract* subclasses: one encapsulating an applicant for undergraduate school, and the other encapsulating an applicant for graduate school. An applicant for undergraduate school has two additional attributes: an SAT score and a GPA. An applicant for graduate school has one additional attribute: the college of origin. It also has a method that returns “from inside” if the college of origin is the same as the college applied to; otherwise, it returns “from outside.” You also need to include a class to test these two classes.
7. Write an *abstract* superclass encapsulating a vacation: A vacation has two attributes: a budget and a destination. It has an *abstract* method returning by how much the vacation is over or under budget. This class has two non--*abstract* subclasses: one encapsulating an all-inclusive vacation, and the other encapsulating a vacation bought piecemeal. An all-inclusive vacation has three additional attributes: a brand (for instance ClubMed®); a rating, expressed as a number of stars; and a price. A piecemeal vacation has two additional attributes: a set of items (hotel, meal, airfare, …), and a set of corresponding costs. You also need to include a class to test these two classes.
8. Write an *abstract* superclass encapsulating a part, with two attributes: the part number, and a budget cost for it. This

class has two non-*abstract* subclasses: one encapsulating a self-manufactured part, and the other encapsulating an outsourced part. A self-manufactured part has a cost and a drawing number; it also has a method returning whether it is over budget or under budget. An outsourced part has a set of suppliers, each with a price for the part. It also has a method to retrieve the lowest-cost supplier for a part and the corresponding cost. You also need to include a class to test these two classes.

9. Write an *abstract* superclass encapsulating a number; this class has one *abstract* void method: *square*. This class has two non-*abstract* subclasses: one encapsulating a rational number, and the other encapsulating a complex number. A rational number is represented by two integers, the numerator and the denominator of the rational number. A complex number is represented by two real numbers, the real part and the complex part of the complex number. You also need to include a class to test these two classes.
10. Define an *abstract UtilityCustomer* class that has one instance variable, an account number, and an *abstract* method, *calculateBill*, that returns the bill amount as a *double*. Also define the constructor, accessors, mutators, and a *toString* method that outputs the account number. Define a *GasCustomer* class that inherits from the *UtilityCustomer* class, and adds an instance variable *cubicMetersUsed* and a constant for the price of gas per cubic meter. Also define an *ElectricCustomer* class that also inherits from the *UtilityCustomer* class and adds a *kWattHourUsed* instance variable along with two constants for the price of electricity per kilowatt hour, plus a flat power delivery fee ($30) that is added to every bill. For each subclass, define the constructor, accessor, and mutator for

its instance variable, a *toString* method, which calls the *toString* method of the *UtilityCustomer* class, and an implementation for the *calculateBill* method. Create a client that instantiates several *GasCustomer* and *ElectricCustomer* objects, adds them to an *ArrayList* of *UtilityCustomer* objects, and then using polymorphism, steps through the *ArrayList*, calling the *calculateBill* method and outputting the bill amount for each customer.

10.10.8 Technical Writing

1. In a large organization, programmers develop a library of classes as they work on various projects. Discuss, in such an environment, how inheritance can be helpful in reusing code and therefore saving time.
2. Other programming languages allow multiple inheritance; that is, a class can inherit from several classes. In Java, a class can extend only one class, but can implement several interfaces. Discuss potential problems that can arise in other programming languages that allow inheritance from multiple classes.

10.10.9 Group Project (for a group of 1, 2, or 3 students)

1. Design and code a program that allows the user to select a drawing style for a graphics program. The styles should be black and white, grayscale, or inverted color. To do this, define a *DrawingStyle* interface with these methods: *setColor* and *setOutline*. Both methods accept a *GraphicsContext* object and a *Color*, convert the color appropriately, and call the *GraphicsContext* methods *setFill* or *setStroke*, respectively, with the new color value.

For each style, define the following classes that implement the *DrawingStyle* interface: *BlackAndWhite*, *Grayscale*, and *InvertColor*.

In the client class, define a *DrawingStyle* instance variable. Prompt the user for the desired style, and assign a reference to an object of the appropriate class that implements the *DrawingStyle* interface. The application should then call *setColor* and *setOutline* to set the fill or stroke color using a reference to the *DrawingStyle* object. Make a drawing of your choice using the selected drawing style.

Hint: Look up the *grayscale* and *invert* methods of the *Color* class. The *BlackAndWhite* class can simply convert any color whose average value for the red, green, and blue components is >= 0.5 to white; otherwise, it converts the color to black.

Description

© Lois GoBe/Shutterstock.

CHAPTER 11

Exceptions and Input/Output Operations

CHAPTER CONTENTS

Introduction
11.1 Simple Exception Handling
11.2 Catching Multiple Exceptions
11.3 Reading Text Files Using *Scanner*
11.4 The *java.io* Package
11.5 Recovering from an Exception
11.6 Writing and Appending to Text Files

    11.6.1 Writing to Text Files
    11.6.2 Appending to Text Files

11.7 Reading Structured Text Files

    11.7.1 Parsing a *String* Using *Scanner*
    11.7.2 Reading Structured Data Using *Scanner*

11.8 **Programming Activity 1: Reading from a Structured Text File**
11.9 Streams
11.10 Reading Formatted Open Data from a Remote

Location

- 11.10.1 Accessing Remote Data
- 11.10.2 JSON Formatting and Parsing
- 11.10.3 Reading, Parsing, Streaming, and Processing Remote Data

11.11 Reading and Writing Objects to a File

- 11.11.1 Writing Objects to Files
- 11.11.2 Reading Objects from Files

11.12 **Programming Activity 2: Reading Objects from a File**
11.13 User-Defined Exceptions
11.14 Chapter Summary
11.15 Exercises, Problems, and Projects

- 11.15.1 Multiple Choice Exercises
- 11.15.2 Reading and Understanding Code
- 11.15.3 Fill In the Code
- 11.15.4 Identifying Errors in Code
- 11.15.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
- 11.15.6 Write a Short Program
- 11.15.7 Programming Projects
- 11.15.8 Technical Writing
- 11.15.9 Group Project

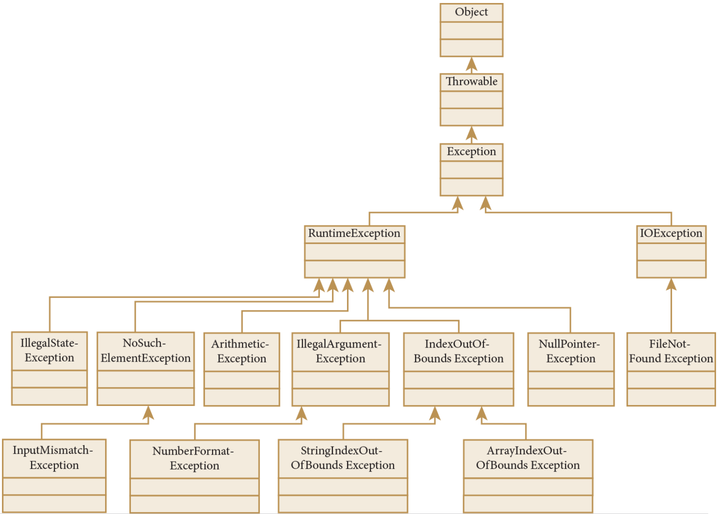
Introduction

Programs often use existing data accumulated by an organization, such as a university, a government, or a corporation. Typically, the volume of data is significant, making data entry through the keyboard impractical.
Furthermore, these large amounts of data typically reside in two types of storage:

- disk files
- databases

Working with databases is beyond the scope of this book. In most of this chapter, we concentrate on reading from and writing to files.

But there is a prerequisite to all this: understanding the concept of exceptions, their associated classes, and exception handling.


Object
Throwable
Exception
RuntimeException
IOException
IllegalState-Exception
NoSuch-ElementException
Arithmetic-Exception
IllegalArgument-Exception
IndexOutOf-Bounds Exception
NullPointer-Exception
FileNot-Found Exception
InputMismatch-Exception
NumberFormat-Exception
StringIndexOut-OfBounds Exception
ArrayIndexOut-OfBounds Exception

```
Message: data.txt (The system cannot find the file specified)

toString( ): java.io.FileNotFoundException: data.txt (The system cannot
find the file specified)

java.io.FileNotFoundException: data.txt (The system cannot find the
file specified)
        at java.base/java.io.FileInputStream.open0(Native Method)
        at java.base/java.io.FileInputStream.open(FileInputStream.java:216)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
        at java.base/java.util.Scanner.<init>(Scanner.java:639)
        at EchoFileData.main(EchoFileData.java:16)
```

11.1 Simple Exception Handling

By now we have discovered that sometimes our program doesn’t work, even though we didn’t get any compiler errors. At run time, logic errors can surface. For example, we might attempt to divide an integer by 0 or try to access the 11th element in a 10-element array. Java is a robust language and does not allow these “illegal” operations to occur unnoticed.

These illegal operations generate **exceptions**. Some exceptions are generated by the Java Virtual Machine, while others are generated by constructors or other methods. For example, a method might generate an exception when it detects an attempted illegal operation or an illegal parameter.

By default, when an exception is generated in an application that does not have a graphical user interface, the program will terminate. In many cases, however, we can attempt to recover from the exception and continue running the program. This is called **handling the exception**. For the programmer to handle an exception, Java provides two tools:

- exception classes
- the *try*, *catch*, and *finally* blocks

The *Exception* class is the superclass of all exception classes, which encapsulate specific exceptions such as integer division by 0, attempting to access an out-of-bounds array index, unsuccessfully converting a *String* to a number, using a *null* object reference to call a method, trying to open a file that does not exist, and others.

Figure 11.1 is an inheritance hierarchy showing only a few of the Java exception classes. The *Exception* class and *RuntimeException* and its subclasses are in the *java.lang* package. The *IOException* class and its subclass, *FileNotFoundException*, are in the *java.io* package.

Figure 11.1 Inheritance Hierarchy for Various *Exception* Classes

Description

We want to avoid situations when an exception occurs and abruptly terminates the execution of our program. Java provides *try* and *catch* blocks to allow us to handle exceptions so that our code can continue to run. We put the code that might generate an exception inside the *try* block, and we put the code to recover from the exception inside a *catch* block. If an exception is thrown by the code inside the *try* block, then execution will jump to the *catch* block, where we provide code to handle that exception. If nothing illegal happens in the *try* block, the code in the *catch* block will be skipped.

The minimum syntax for a *try* and *catch* block is as follows:

```
try
{
  // code that might generate an exception
}
catch ( ExceptionClass exceptionObjRef )
{
  // code to recover from the exception
}
```

The curly braces are required for both the *try* body and the *catch* body even if the bodies have only one statement, or even no statements.

COMMON ERROR TRAP

Omitting curly braces around the *try* and *catch* blocks will generate a compiler error. A *catch* clause listing several *Exception* classes as parameters will also generate a compiler error. Use multiple *catch* blocks instead.

Any variable defined within the *try* block is local to that block; that is, the variable cannot be referenced after the *try* block, not even in the *catch* block. Note that the *ExceptionClass* parameter of the *catch* clause specifies one and only one *ExceptionClass*. Listing zero or two or more *ExceptionClasses* in the *catch* clause will generate a compiler error.

Java distinguishes between two types of exceptions:

- unchecked, those that are subclasses of *Error* or *RuntimeException*
- checked, any other exception class

An **unchecked exception**, such as an *ArithmeticException* caused by attempting integer division by 0, a *NumberFormatException*, or a *NullPointerException*, does not have to be handled with a *try* and *catch* block. In other words, if we omit the *try* and *catch* blocks, our code will compile without an error. If one of these unchecked exceptions is generated at run time, however, the JVM will catch it and print output similar to that shown in Figure 11.2 (discussed later in this section).

Figure 11.2 Output of Example 11.1

Description

Code that could generate a **checked exception**, such as an *IOException*, must be coded within a *try* block. Optionally, the method header can acknowledge that the checked exception could occur by using a *throws* clause. This is required; otherwise, the program will not compile. Thus, when we perform I/O on a file, our code must deal with a potential *IOException*.

In the *catch* block, we can use the *Exception* parameter as an object reference to get more information about what caused the exception. Table 11.1 shows three methods inherited by the *Exception* classes.

Useful Methods of Exception Classes

Methods of Exception Classes

Return value	**Method name and argument list**
`String`	`getMessage( )` returns a message indicating the cause of the exception
`String`	`toString( )` returns a *String* containing the exception class name and a message indicating the cause of the exception
`void`	`printStackTrace( )` prints the line number of the code that caused the exception, along with the sequence of method calls leading up to the exception

Description

We demonstrate these three methods, as well as how to use *try* and *catch* blocks to detect and handle an exception if it occurs, in **Example 11.1**. When we want to read data from a file, we first must open the file for reading. At line 15, we declare and instantiate a *File* object for a file named *data.txt* located in the current folder. At line 16, we declare and instantiate a

Scanner object so we can read from the input stream represented by that *File* object. The *Scanner* constructor, whose API is shown below, *throws* a *FileNotFoundException* if the file is not found.

Example 11.1 Using *try* and *catch* Blocks

```
 1 /* Reading a Text File
 2    Anderson, Franceschi
 3 */
 4
 5 import java.io.File;
 6 import java.io.FileNotFoundException;
 7 import java.util.Scanner;
 8
 9 public class EchoFileData
10 {
11   public static void main( String [ ] args )
12   {
13     try
14     {
15       File inputFile = new File( "data.txt" );
16       Scanner file = new Scanner( inputFile );
17
18       while ( file.hasNextInt( ) )
19       {
20         int number = file.nextInt( );
21         System.out.println( number );
22       }
23     }
24     catch ( FileNotFoundException fnfe )
25     {
26       System.out.println( "Message: " + fnfe.getMessage( ) );
27       System.out.println( "\ntoString( ): " + fnfe + "\n" );
28       fnfe.printStackTrace( );
29     }
30   }
31 }
```

Description

```
public Scanner( File source ) throws FileNotFoundException
```

Description

The *FileNotFoundException* is a checked exception. Thus, we need to place the code that can generate that exception (line 16) inside a *try* block (lines 13–23). At lines 24–29, we use a *catch* block to handle the exception if it occurs. Note that there are other exceptions that can be thrown in this program.

The *File* constructor, as well the *hasNextInt* and *nextInt* methods of the *Scanner* class, can throw exceptions; however, all those exceptions are unchecked exceptions and do not require the use of *try/catch* blocks.

When the program runs, if there is no file named *data.txt* in the current folder, the *try* block stops executing at line 16. Next, the *catch* block is executed, and we print the values from the *getMessage*, *toString*, and *printStackTrace* methods of the -*FileNotFoundException* class. Figure 11.2 shows the output of Example 11.1 when there is no file named *data.txt* in the current folder. As we can see, the only difference between the return values from the *getMessage* and *toString* methods is that the *toString* method returns the exception class name, as well as the message. The output of the *printStackTrace* method may look familiar. It is similar to a message that the JVM prints when an *ArithmeticException*, *StringIndexOutOfBoundsException*, or *ArrayIndexOutOfBoundsException* occurs when we do not have *try* and *catch* blocks.

11.2 Catching Multiple Exceptions

If the code in the *try* block might generate multiple types of exceptions, we can provide multiple *catch* blocks, one for each possible exception. When an exception is generated, the JVM searches the *catch* blocks in order. The first *catch* block with a parameter that matches the exception thrown will execute; any remaining *catch* blocks will be skipped.

Remember that subclass objects are also objects of their superclasses, so an exception will match any *catch* block for an exception that names any of its superclasses. For example, a *NumberFormatException* will match a *catch* block with a *RuntimeException* parameter, and all exceptions will match a *catch* block with an *Exception* parameter. Therefore, when coding several *catch* blocks, put the *catch* blocks for the specialized exceptions first, followed by more general exceptions.

SOFTWARE ENGINEERING TIP

Arrange *catch* blocks to handle the more specialized exceptions first, followed by more general exceptions.

Furthermore, after a *try* block and its associated *catch* blocks, we may optionally add a *finally* block, which will always be executed, whether an exception occurred or not. In the *finally* block, we can include some clean-up code. We will demonstrate a *finally* block when we read from a file later in this chapter.

Here is the syntax for using a *try* block, several *catch* blocks, and a *finally* block:

```
try
{
   // code that might generate an exception
}
catch ( Exception1Class e1 )
{
   // code to handle an Exception1Class exception
}
...
catch ( ExceptionNClass eN )
{
   // code to handle an ExceptionNClass exception
}
finally
{
   // code to execute regardless of whether an exception occurs
}
```

Description

Again, the curly braces around the various blocks are required, whether these blocks contain zero, one, or more statements.

SOFTWARE ENGINEERING TIP

Whenever possible, use a simple *if/else* statement to detect an unchecked exception, rather than *try* and *catch* blocks. This will improve the performance of your code.

Having provided several examples of exceptions, we must also consider this: Not every problem needs to be addressed by generating an exception. As a matter of fact, generating and handling exceptions considerably slows down execution of our code due to the processing overhead. Often, for example when using Java’s I/O classes, we will have no choice but to use *try* and *catch* blocks. Sometimes, however, we can use a simple *if/else* statement instead of *try* and *catch* blocks. For example, we can test if the value of the divisor is not zero before attempting a division, rather than placing the code in a *try* block and catching an *ArithmeticException*.

How do we know if a constructor or a method *throws* an exception and what type of exception it *throws*? As always, our best source of information is the Java Class Library on Oracle’s Java website. After we have identified a constructor or a method that we would like to use, we simply look at its description in order to determine whether it *throws* any exceptions, and, if so, which ones.

REFERENCE POINT

Consult the Oracle Java Class Library to see if a constructor or a method *throws* an exception and, if so, what type of exception.

```
The Matrix
136
Finding Nemo
100
Titanic
194
Casablanca
102
Gone With the Wind
220
```

```
The Matrix, 136 minutes
Finding Nemo, 100 minutes
Titanic, 194 minutes
Casablanca, 102 minutes
Gone With the Wind, 220 minutes
```

```
Unable to find movies.txt, exiting
```

11.3 Reading Text Files Using *Scanner*

Java supports two file types, text and binary. In text files, data is stored as characters; in binary files, data is stored as raw bytes. Different classes are used for writing and reading each file type. The type of a file is determined when the file is written and depends on which classes were used to write to the file. Thus, to read from an existing file, we must know the file's type in order to select the appropriate classes.

In this section, we concentrate on text files.

As we have seen from Example 11.1, the *Scanner* class can be used to read from a text file. If the file is not found, the *Scanner* constructor throws a *FileNotFoundException*. More exceptions can be thrown by *Scanner* methods while reading the file.

Table 11.2 shows some constructors of the *Scanner* and *File* classes and methods of the *Scanner* class for reading from a file, as well as the exceptions that each method can throw.

Useful Methods of the *Scanner* and *File* Classes for Reading from a File

File Constructor	**Exceptions Thrown**
`File( String pathname )` constructs a *File* object with the *pathname* file name so that the file name is platform-independent	`NullPointerException`

***Scanner* Constructor and Methods**	**Exceptions Thrown**
`Scanner( File file )` constructs a *Scanner* object for reading from a file	`FileNotFoundException`

Return Value	**Method Name and Argument List**	**Exceptions Thrown**
`boolean`	`hasNext( )` returns *true* if there is another token in the input stream	`Both methods throw an IllegalStateExceptio`
`boolean`	`hasNextLine( )` returns *true* if there is another line or line separator in the input stream	
`boolean`	`hasNextlnt( )`	`All these methods throw an IllegalStateExceptio`
`boolean`	`hasNextDouble( )`	
`boolean`	`hasNextFloat( )` ... returns *true* if there is another token in the input stream of the specified data type	
`String`	`next( )` returns the next token	`Both methods throw`

	in the input stream as a *String*	`NoSuchElementExceptio` `IllegalStateExceptio`
`String`	`nextLine( )` returns the remainder of the line as a *String*. Positions the pointer to the next line.	

`int`	`nextInt( )`	`All these methods throw InputMismatchExceptio`
		`NoSuchElementExceptio`
		`IllegalStateExceptio`

`double`	`nextDouble( )`	
`float`	`nextFloat( )` ... returns the next token in the input stream as the specified data type	
`void`	`close( )` releases the resources associated with an open input stream	None

Description

In **Example 11.2**, we read a text file named *movies.txt*, which contains titles of classic movies and the movies' running times in minutes. Each movie occupies two lines in the file: the title on one line and the running time on the following line. Each line in the file, including the last line, is terminated by a *newline* character. In this example, we simply read the movie titles and running times and echo them to the console. However, we could read this file to find the longest or shortest movie, or to instantiate *Movie* objects to be stored in an *ArrayList* or other data structure for further processing.

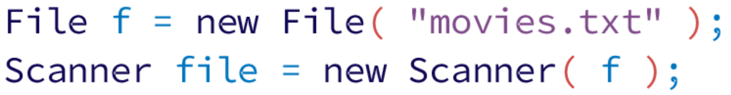
Example 11.2 Reading from a Text File

```
 1 /** Classic Movies - Reading from a text file
 2 *   Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6 import java.io.File;
 7 import java.util.NoSuchElementException;
 8 import java.io.FileNotFoundException;
 9
10 public class ClassicMovies
11 {
12  public static void main( String [ ] args )
13  {
14   try ( Scanner file = new Scanner( new File( "movies.txt" ) ) )
15   {
16    while ( file.hasNext( ) ) // test for the end of the file
17    {
18     String movieTitle = file.nextLine( );
19
20     if ( ! file.hasNextInt( ) )
21     {
22      System.out.println( "Invalid file format" );
23      String invalidData = file.nextLine( ); // skip the line
24     }
25     else
26     {
27      int runningTime = file.nextInt( );
28      String newLine = file.nextLine( ); // read newline character
29      System.out.println( movieTitle + ", "
30               + runningTime + " minutes" );
31     }
32    }
33   }
34
35   catch ( FileNotFoundException fnfe )
36   {
37    System.out.println( "Unable to find movies.txt, exiting" );
38   }
39
40   catch ( NoSuchElementException nsee )
41   {
42    System.out.println( "Attempt to read past the end of the file" );
43   }
44  }
45 }
```

Description

Lines 7–8 import the *NoSuchElementException* and *FileNotFoundException* classes. The *IllegalStateException* class does not need to be imported because it is in the *java.lang* package. Line 14 instantiates a *Scanner* object from a *File* object, passing the name of the file to be read, *movies.txt*, as the argument to the *File* constructor.

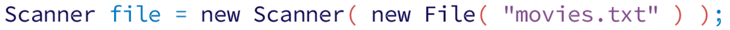
The *File* object is used only as the argument of the *Scanner* constructor; it is not used anywhere else in the program. In this case, instead of creating a *File* object reference, many programmers prefer to use an anonymous *File* object as the argument of the *Scanner* constructor. In other words, instead of writing the following two statements:



```
File f = new File( "movies.txt" );
Scanner file = new Scanner( f );
```

Description

we use the following single statement:



```
Scanner file = new Scanner( new File( "movies.txt" ) );
```

Description

It is a matter of preference as to which code we use.

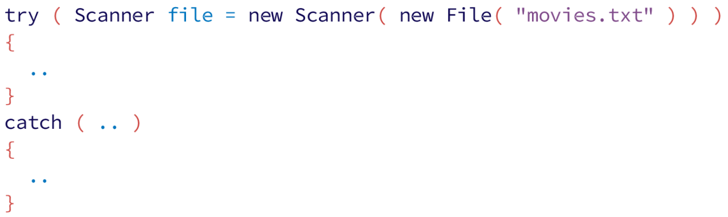
When we associate a file with an input stream or output stream, we are **opening the file**. As we read the data in a file, a **file pointer** keeps track of the next data to read. When we open the file, the file pointer is set to the beginning of the file. When we are finished with a file, we can call the *close* method (shown in Table 11.2) to release the resources associated with the file. Calling the *close* method is optional. When the program finishes executing, all of its resources are released, including the resources of any unclosed files. Nevertheless, it is good practice, in general, to call the *close* method, especially if we will be opening a number of files (or opening the same file multiple times). In contrast, the standard input stream (*System.in*), the standard output stream (*System.out*), and the standard error stream (*System.err*) are open when the program begins. They are intended to stay open and should not be closed.

SOFTWARE ENGINEERING TIP

Either use a *try*-with-resources statement or close files when you have finished processing their data. Do not close the standard input, output, or error devices.

At lines 14–33, we use a ***try*-with-resources** statement. A **resource** is an object reference, for example a *Scanner*, that should be closed after we no longer need it. A *try*-with-resources is a statement that declares one or more resources. Using a *try*-with-resources statement guarantees that the resource will be automatically closed at the end of the *try* block. If we did not use a *try*-with-resources statement at line 14, we would need to close the *Scanner* in the *finally* block to guarantee that it is closed. Closing the *Scanner* as the last statement of the *try* block would not guarantee that the *Scanner* is closed because if an exception occurs, the *try* block will not finish executing.

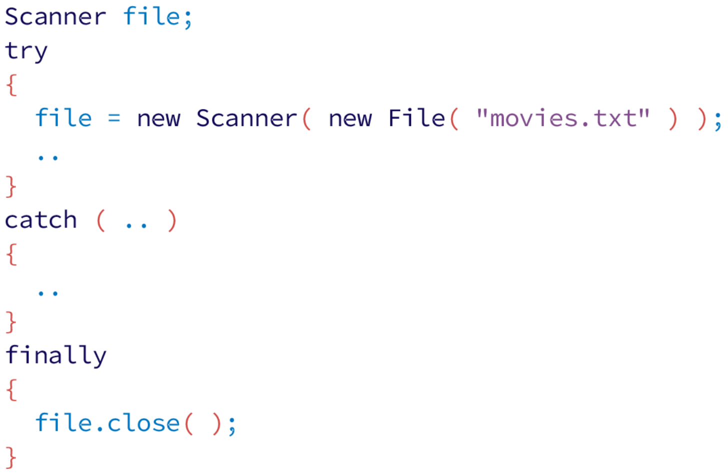
Thus, the following code:



```
try ( Scanner file = new Scanner( new File( "movies.txt" ) ) )
{
  ..
}
catch ( .. )
{
  ..
}
```

Description

is equivalent to:



```
Scanner file;
try
{
  file = new Scanner( new File( "movies.txt" ) );
  ..
}
catch ( .. )
{
  ..
}
finally
{
  file.close( );
}
```

Description

The *Scanner* constructor throws a *FileNotFoundException* if the file does not exist. We catch this exception at lines 35–38.

Since the program cannot continue without the file being open, we print a message to the user that we were not able to find the file and that the program is exiting. Thus, instead of facing a *FileNotFoundException* with a stack trace, the user now sees a friendly message explaining the problem.

If no exception is thrown, we begin to read the file. The *while* loop condition, at line 16, uses the *Scanner hasNext* method to test whether the file pointer has reached the end of the file. If not, *hasNext* returns *true*, and we execute the *while* loop body. Since the movie title could contain more than one word, we use the *nextLine* method to read the entire line (line 18).

We then test whether the next token in the file is an *int* (line 20). If not, then the running time is not a whole number. We output a message that the file format is invalid, then skip the line by using the *nextLine* method to read whatever data is on that line into the *String invalidData* (lines 22–23). Another option is to use a *try/catch* block to read the running time. With that option, calling the *nextInt* method to read the invalid data would generate an *InputMismatchException*, which we would need to handle. Given that exception handling results in additional processing overhead, our using the *hasNextInt* method of the *Scanner* class to test whether the file format is valid makes the code more efficient.

If the next token is an *int*, we read the running time using the *nextInt* method (line 27). At this point, it is worthwhile to discuss the difference between the *nextLine* method and the other *Scanner next…* methods (such as *next/nextInt/nextDouble*, etc.). The *nextLine* method reads any part of the line that has not already been read, including leading and trailing white space (spaces, tabs, and the *newline* character) and moves the file pointer to the next line. The *nextLine* method does not include the *newline* character in its returned *String*. In contrast, the *nextInt* method and its *next…* method counterparts skip leading white space, then read the next token on the line, stopping when trailing white space is encountered. That means that after we have read the running time using the *nextInt* method, the *newline* character is still in the input stream. If we then attempt to read the next movie title using the *nextLine* method, the *nextLine* method would instead read just the *newline* character that follows the running time and return an empty *String*. So on line 28, we insert an additional call to the *nextLine* method to remove that *newline* character from the input stream.

On lines 29–30, we echo the title and running time to the console.

We continue looping through the file until there is no more data to read.

An attempt to access a file after it has been closed generates an *IllegalStateException*. Because we use a *try*-with-resources statement, the *Scanner* will be closed after the last *catch* block. Thus, we cannot access the file after it has been closed and do not need to *catch* an *IllegalStateException*. If we attempt to read beyond the end of the file, the *nextLine* or *nextInt* methods will throw a *NoSuchElementException*. We catch this exception on lines 40–43, and output a message explaining the cause of the exception.

Let's assume the file *movies.txt* contains the data shown in **Figure 11.3**. When the program in Example 11.2 runs, it will produce the output shown in **Figure 11.4** if the file is found, and the output in **Figure 11.5** if the file is not found.

Figure 11.3 Contents of *movies.txt*

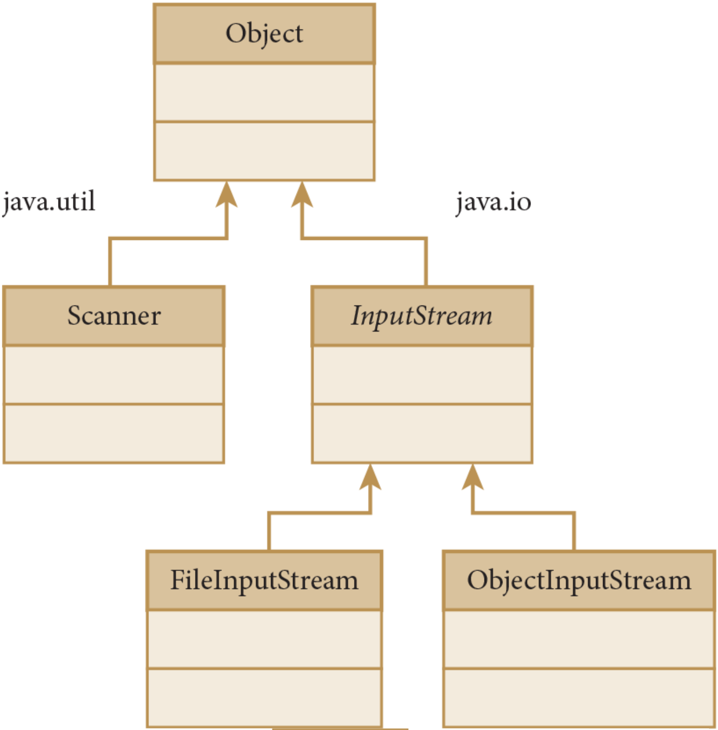
Description

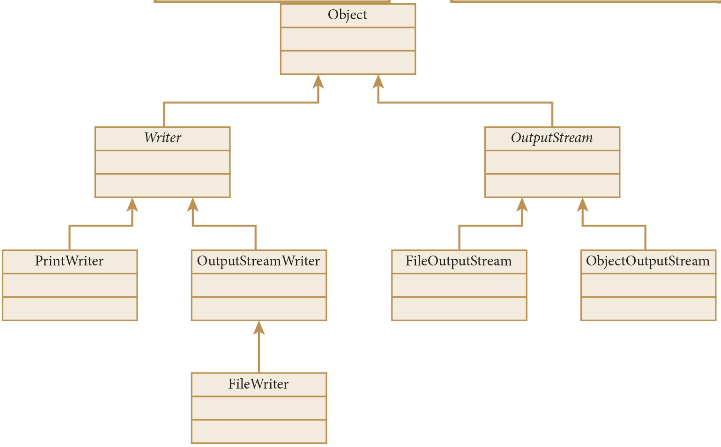
Figure 11.4 Output of Example 11.2 When File Is Found

Description

Figure 11.5 Output of Example 11.2 When File Is Not Found

Description


Object
java.util
java.io
Scanner
InputStream
FileInputStream
ObjectInputStream


Object
Writer
OutputStream
PrintWriter
OutputStreamWriter
FileOutputStream
ObjectOutputStream
FileWriter

11.4 The *java.io* Package

In addition to the *Scanner* class in the *java.util* package, Java provides a number of classes in the *java.io* package for reading from files and for writing to files. We will use only a few of those classes here. **Table 11.3** describes a group of classes designed for data input.

Selected Input Classes

Input Classes

Class	**Description**
`Scanner`	Class to read and parse characters in text files
`InputStream`	*Abstract* superclass representing an input stream of raw bytes
`FileInputStream`	Input stream to read raw bytes of data from files
`ObjectInputStream`	Class to read/recover objects from a file written using *ObjectOutputStream*

Description

Figure 11.6 shows an inheritance hierarchy for the Java classes described in Table 11.3.

Figure 11.6 The Inheritance Hierarchy for Input Classes

Description

Table 11.4 describes a group of classes designed for data output, and **Figure 11.7** shows an inheritance hierarchy for Java output classes in Table 11.4.

Selected Output Classes

Output Classes

Class	**Description**
`Writer`	*Abstract* superclass for writing characters to output streams
`OutputStream`	*Abstract* superclass representing

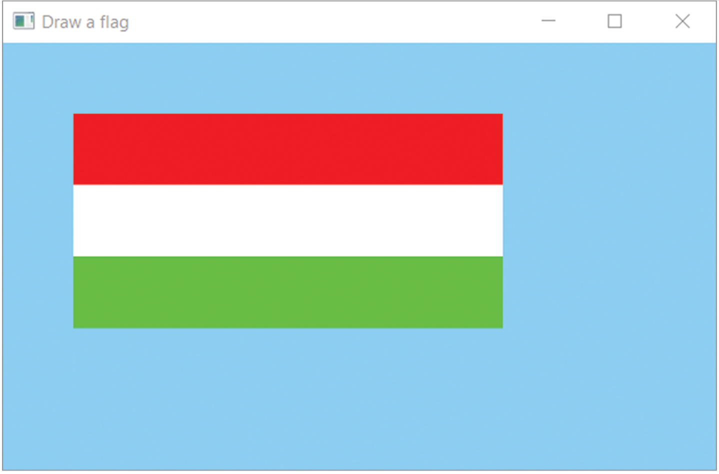
	an output stream of raw bytes
`FileWriter`	Convenience class for writing characters to files
`PrintWriter`	Convenience class to output basic data types, *Strings*, and objects
`FileOutputStream`	Output stream to write raw bytes of data to files
`ObjectOutputStream`	Class to write objects to a file

Description

Figure 11.7 The Inheritance Hierarchy for Output Classes

Description

```
300
15000 1.0 0.0 0.0
15000 1.0 1.0 1.0
15000 0.0 1.0 0.0
```



11.5 Recovering from an Exception

In the next example, we read data from a file and use that data to draw a flag. **Figure 11.8** shows the contents of the *HungarianFlag.txt* file. We also show how to catch and recover from an exception; that is, how to enable our program to continue to run when an exception occurs. The data is organized in a similar manner, but it is simpler than what is used by the GIF run-length encoding algorithm:

Figure 11.8 The *HungarianFlag.txt* File

Description

- The first line is an integer storing the width of the flag.
- Each line thereafter contains four numbers. The first number, an integer, stores the number of consecutive pixels (as we go left to right on the flag image) of the same color; the next three numbers store the red, green, and blue coefficients of that color as *doubles* between 0 and 1.

The width of the Hungarian flag represented by our file is 300, it has 15,000 consecutive red pixels (i.e., 15,000 divided by 300, which is equal to 50, red lines), then 15,000 consecutive white pixels (i.e., 50 white lines), and 15,000 consecutive green pixels (i.e., 50 green lines).

Example 11.3 shows the *GIFDecoding* class. Its constructor accepts an already instantiated *Scanner* parameter and reads the data in the associated file into three instance variables: *width*, an *int*; *lengthList*, an *ArrayList* of *Integers*; and *colorList*, an *ArrayList* of *Color* references. The *lengthList* and *colorList* are parallel *ArrayLists*; that is, the color stored at a given index in *colorList* applies to the number of pixels stored in *lengthList* at that same index.

Example 11.3 The *GIFDecoding* Class

```java
/* GIF decoding class
   Anderson, Franceschi
*/

import java.io.*;
import java.util.*;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;

public class GIFDecoding
{
  private int width; // the width of the GIF

  // ArrayList instance variables:
  // colorList stores Color references
  // lengthList stores integers, the length of each run
  private ArrayList<Color> colorList;
  private ArrayList<Integer> lengthList;

  /* GIFDecoding constructor
   * @param file a Scanner, already open for reading a GIF file
   * initializes width with the first int in file
   * fills the two ArrayLists with the
   * ints and colors stored in file
  */
  public GIFDecoding( Scanner file )
  {
    colorList = new ArrayList<Color>( );
    lengthList = new ArrayList<Integer>( );

    try
    {
      width = file.nextInt( );
      int length;
      double red, green, blue;

      while ( file.hasNext( ) )
      {
        length = file.nextInt( );
        lengthList.add( length );

        red = file.nextDouble( );
        green = file.nextDouble( );
        blue = file.nextDouble( );
        colorList.add( Color.color( red, green, blue ) );
      }
    }
    catch ( InputMismatchException ime )
    {
      System.out.println( "Invalid file format; exiting" );
    }
    catch ( Exception e )
    {
      System.out.println( e.getMessage( ) );
    }
    finally
    {
      file.close( );
    }
  }

  /*
   * drawGIF method
   * @param gc a GraphicsContext reference, the graphics context
   * @param x an int, the drawing's upper left corner x coordinate
   * @param y an int, the drawing's upper left corner y coordinate
  */
  public void drawGIF( GraphicsContext gc, int x, int y  )
  {
    // draw the gif
    Color currentColor;
    int index = 0;
    int deltaX = 0;
    for ( int length : lengthList )
    {
      currentColor = colorList.get( index );
      gc.setStroke( currentColor );

      // draw one or more lines using length pixels with currentCo
      while ( length >= width - deltaX )
      {
        gc.strokeLine( x + deltaX, y, x + width, y );
        y++;
        length = length - ( width - deltaX );
        deltaX = 0;
      }

      if ( length > 0 )
        gc.strokeLine( x + deltaX, y, x + deltaX + length, y );

      deltaX = deltaX + length;
      index++;
    }
  }
}
```

Description

Inside the *try* block, we first read the width of the flag at line 33 and assign it to *width*. At lines 37–46, we loop through the remaining lines of the file, reading the number of pixels (line 39) and their three color components (lines 42–44). With the three color components, we create a *Color* object and add it to *colorList* at line 45; we add the number of consecutive pixels of that color to *lengthList* at line 40.

We catch an *InputMismatchException* at lines 48–51, in case the data inside the file is not what we expect, and exit. A *NoSuchElementException* and an *IllegalStateException* can also be thrown by the *nextInt* and *nextDouble* methods. We catch them both in the generic *Exception catch* block at lines 52–55. We close the *Scanner* in the *finally* block (lines 56–59).

The *drawGIF* method (lines 62–94) accepts a *GraphicsContext* parameter and uses it to draw the GIF. The upper-left coordinates of the GIF are defined by the *x* and *y* parameters passed to *drawGIF*. At lines 74–93, we loop through all the lengths stored in *lengthList*. At line 76, we retrieve the corresponding color for the current pixels and set the stroke color of the *GraphicsContext* to that color at line 77. Because the number of consecutive pixels of the same color can be larger than the width of the drawing, several consecutive lines could have the same color. Thus, we draw the GIF one line at a time using a *while* loop (lines 79–86). After the loop, we draw a line with the current color if there are pixels left over (lines 88–89).

In **Example 11.4**, we ask the user to input a file name, instantiate a *Scanner* to read the file, create and instantiate a *GIFDecoding* object, then call the *drawGIF* method to draw the flag. If there is no file in the current folder with the name entered by the user or if the user mistypes the file name, a *FileNotFoundException* is thrown. Instead of terminating the program, we use the *try* and *catch* blocks to recover from erroneous user input. We use a *do/while* loop at lines 25–40 that keeps asking the user for a file name as long as no file with that name can be found in the current folder. At line 23, we declare a *boolean* variable, *fileFound*, initialized to *false*. As long as *fileFound* is *false* (line 40), we keep entering the loop and asking the user for a file name. If the user enters an invalid file name, our code instantiating the *Scanner file* (line 33) will throw a *FileNotFoundException*. We will then skip line 34 and execute the catch block (lines 36–39). Thus, *fileFound* will remain *false* and our loop condition will remain *true*. When the user enters a valid file name, *file* will be successfully instantiated and no exception will be thrown. Line 34 will execute, changing *fileFound* to *true* and we will then exit the loop.

Example 11.4 Reading a File and Drawing the Corresponding GIF

```
/* Drawing a GIF
   Anderson, Franceschi
*/

import java.io.*;
import java.util.Scanner;
import javafx.application.Application;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class DrawAGIF extends Application
{
  final int WIDTH = 500;
  final int HEIGHT = 300;

  @Override
  public void start( Stage stage )
  {
    Scanner scan = new Scanner( System.in );
    Scanner file = null;
    String filename = "";
    boolean fileFound = false;

    do
    {
      try
      {
        System.out.print( "Enter the name of the file > " );
        filename = scan.next( );

        File inputFile = new File( filename );
        file = new Scanner( inputFile );
        fileFound = true;
      }
      catch ( FileNotFoundException fnf )
      {
        System.out.println( "Unable to find " + filename );
      }
    } while ( !fileFound );

    // set up window title and size
    GraphicsContext gc = JIGraphicsUtility.setUpGraphics(
            stage, "Draw a flag", WIDTH, HEIGHT );
    gc.setFill( Color.LIGHTSKYBLUE );
    gc.fillRect( 0, 0, WIDTH, HEIGHT );

    GIFDecoding flag = new GIFDecoding( file );
    flag.drawGIF( gc, 50, 50 );
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description

COMMON ERROR TRAP

Failing to initialize a variable that is assigned a value in a *try* block, then accessed after the *try* block, will generate a compiler error.

Notice that we declare and initialize the variables *file* and *filename* at lines 21 and 22 before we enter the *try* block. If we do not initialize them and then try to access them in the *catch* block or after the *try*/*catch* blocks, we will receive the following compiler error (shown for *filename*):

```
variable filename might not have been initialized
```

At lines 42–46, we create a *GraphicsContext*, instantiate a *GIFDecoding* object using *file* at line 48, and draw the flag at line 49.

When we run and enter *HungarianFlag.txt* at the prompt, the window shown in **Figure 11.9** appears.

Figure 11.9 Output of Example 11.4

Description

SOFTWARE ENGINEERING TIP

Write code to catch and handle exceptions generated by invalid user input. Although the methods of the *Exception* class are good debugging tools, they are not necessarily appropriate to use in the final version of a program. Try to write user-friendly error messages.

```
30
10 0.0 1.0 0.0
10 1.0 1.0 1.0
10 1.0 0.0 0.0
10 0.0 1.0 0.0
10 1.0 1.0 1.0
10 1.0 0.0 0.0
10 0.0 1.0 0.0
10 1.0 1.0 1.0
10 1.0 0.0 0.0
10 0.0 1.0 0.0
10 1.0 1.0 1.0
10 1.0 0.0 0.0
10 0.0 1.0 0.0
10 1.0 1.0 1.0
10 1.0 0.0 0.0
10 0.0 1.0 0.0
10 1.0 1.0 1.0
10 1.0 0.0 0.0
10 0.0 1.0 0.0
10 1.0 1.0 1.0
10 1.0 0.0 0.0
10 0.0 1.0 0.0
10 1.0 1.0 1.0
10 1.0 0.0 0.0
10 0.0 1.0 0.0
10 1.0 1.0 1.0
10 1.0 0.0 0.0
10 0.0 1.0 0.0
10 1.0 1.0 1.0
10 1.0 0.0 0.0
```

11.6 Writing and Appending to Text Files

In the previous sections, we learned how to read data from a text file. But how did the data get into the file in the first place? It could be that someone put the data into the file using a text editor, such as Notepad in Windows, or TextEdit in MacOS. Typing data into a file is convenient when the amount of data is small. But very often files contain a significant amount of data, typically written to the file by a computer program. For instance, a web server writes to log files to keep track of the visitors accessing the website, how the visitors got to the website, the time they arrived, etc. If the web server comes under attack from a hacker, these log files can be consulted to determine where the hacker came from, who the hacker was, and other information.

In this section, we will learn how to write to a text file. But before going into the details, we must distinguish among several situations:

- Creating/writing to a new file, that is, the file does not exist.
- Writing to an existing file and replacing the contents of the file with new data.
- Writing to an existing file, but keeping the contents of the file and adding data at the end of the file. This is called **appending** to the file.

Java provides us with the necessary tools to perform all the preceding actions. **Table 11.5** summarizes what will happen, depending on the action we perform and whether the file already exists.

Writing or Appending to an Existing or New File

Operation	If the File Exists …	If the File Does Not Exist …
write	the current contents of the file are deleted, and writing starts at the beginning of the file	the file is created and writing starts at the beginning of the file
append	data is added to the end of the file, keeping the existing contents	the file is created and writing starts at the beginning of the file

Description

11.6.1 Writing to Text Files

The data that we write to files can be primitive data types, such as *ints*, *doubles*, or *booleans*, or even objects, such as *Strings*.

FileWriter, a subclass of the *Writer* class, is designed to write characters to a file. It has several constructors, one of which takes a file name and a mode as its two arguments. The *boolean mode* variable specifies whether we are writing (*false*) or appending (*true*) to the file.

The *PrintWriter* class is designed for converting basic data types to characters and writing them to a text file. The *PrintWriter* class provides *print* and *println* methods for all primitive data types, as well as for *Strings* and objects, that is, it calls the *toString* method of the object's class. The *print* method writes the argument value to the file, whereas the *println* method writes the argument value to the file followed by a *newline* character. The constructors and method APIs are shown in **Table 11.6**.

Useful Classes, Constructors, and Methods for Writing to a Text File

Classes, Constructors, and Methods for Writing to a Text File

Class	**Constructors**	**Exceptions thrown**
`FileWriter`	`FileWriter( String filename, boolean mode )` constructs a *FileWriter* object from a *String* representing the name of a file; if *mode* is *false*, we will write to the file; if *mode* is *true*, we will append to the file	`IOException`
`PrintWriter`	`PrintWriter( Writer wr )` constructs a *PrintWriter* object from the *Writer* object *wr*	None
	Method APIs	
`PrintWriter`	`void print( int i )`	None

	`void print( double d )`
	`void print( char c )`
	`void print( boolean b )`
	`...`
	`void println( int i )`
	`void println( double d )`
	`...`
	writes the argument to a text file	

`PrintWriter`	`void close( )` releases the resources associated with the *PrintWriter* object	None

Description

Example 11.5 shows how we can use the *FileWriter* and *PrintWriter* classes to write *Strings* and primitive data types to a text file named *ItalianFlag.txt*. We write data for the Italian flag that can later be read and drawn using the *DrawAGIF* example.

Example 11.5 Writing *Strings* and Primitive Data Types to a Text File

```
 1 /*  Writes a GIF encoding to a file
 2     Anderson, Franceschi
 3 */
 4
 5 import java.io.*;
 6
 7 public class WriteFlagToFile
 8 {
 9   private static final int HEIGHT = 10;
10
11   public static void main( String [ ] args )
12   {
13     try
14     {
15       FileWriter fw = new FileWriter( "ItalianFlag.txt", false );
16       PrintWriter pw = new PrintWriter( fw );
17       pw.println( 30 ); // width of GIF
18       // write HEIGHT lines of color and pixels data
19       for ( int i = 0; i < HEIGHT; i++ )
20       {
21         pw.print( 10 ); // 10 green pixels
22         pw.print( ' ' ); // white space character
23         pw.print( 0.0 ); // red color component
24         pw.println( " 1.0 0.0" ); // green and blue color components
25
26         pw.println( "10 1.0 1.0 1.0" ); // 10 white pixels
27         pw.println( "10 1.0 0.0 0.0" ); // 10 red pixels
28       }
29
30       pw.close( );
31     }
32
33     catch ( IOException ioe )
34     {
35       System.out.println( ioe.getMessage( ) );
36     }
37   }
38 }
```

Description

Line 15 instantiates a *FileWriter* object to write to the file *ItalianFlag.txt*. The *FileWriter* constructor could throw an *IOException*. Our code catches that exception at lines 33–36. This is the only *catch* block because the methods of the *PrintWriter* class do not throw exceptions.

Line 16 instantiates a *PrintWriter* object, which we will use to write to the file. At lines 17–28, using the *PrintWriter* object *pw*, we call the methods *print* and *println*, passing various *String* and primitive data type arguments (*int*, *char*, and *double*) to be written to the file. When we want a *newline* character appended to the output, we call *println*, rather than *print*.

On line 30, we close the file. Although calling the *close* method is optional when we are reading a file, it is essential to call the *close* method when we are writing to a file. Closing the file flushes any buffered data to the output file. If we omit calling the *close* method, we may find that the file is empty when the program ends.

After this program is executed, the file *ItalianFlag.txt* will contain the data shown in **Figure 11.10**. Because we wrote all the output to the file *ItalianFlag.txt*, there is no output to the console.

Figure 11.10 The *ItalianFlag.txt* File After Running Example 11.5

Description

11.6.2 Appending to Text Files

Appending text to a file is similar to writing text; the only difference is that the second argument of the *FileWriter* constructor is *true*, instead of *false*.

Example 11.6 shows how the *FileWriter* and *PrintWriter* classes can be used in a Java program to append text to our file named *ItalianFlag.txt*.

Example 11.6 Appending to a Text File

```java
/*  Appends to a file
    Anderson, Franceschi
*/

import java.io.*;
import java.util.*;

public class AppendToFlagFile
{
  static final int PIXELS = 10;

  public static void main( String [ ] args )
  {
    int lines = 0;
    Scanner scan = new Scanner( System.in );
    boolean goodInput = false;

    do
    {
      try
      {
        // prompt for input; expected value is an int
        System.out.print( "Enter a number of lines (20 to 50) "
                + "to append > " );
        lines = scan.nextInt( );
        if ( lines >= 20 && lines <= 50 )
          goodInput = true;
      }

      catch ( InputMismatchException ime )
      {
        // consume invalid data left in input stream
        String garbage = scan.nextLine( );
        System.out.print( "You did not enter an integer; "
                + "please enter an integer > " );
      }
    } while ( !goodInput );

    try
    {
      FileWriter fw = new FileWriter( "ItalianFlag.txt", true );
      PrintWriter pw = new PrintWriter( fw );
      // append lines of color and pixels data for this flag
      for ( int i = 0; i < lines; i++ )
      {
        pw.println( PIXELS + " 0.0 1.0 0.0" ); // PIXELS green pixels
        pw.println( PIXELS + " 1.0 1.0 1.0" ); // PIXELS white pixels
        pw.println( PIXELS + " 1.0 0.0 0.0" ); // PIXELS red pixels
      }
      pw.close( );
    }

    catch ( IOException ioe )
    {
      System.out.println( ioe.getMessage( ) );
    }
  }
}
```

Description

Example 11.6 is similar to Example 11.5. The major difference is that when we instantiate the *FileWriter* object (line 41), the second argument is *true*, which means that we will append to the file *ItalianFlag.txt*. If the file *ItalianFlag.txt* exists, we will start writing at the end of its current contents, whereas if the file does not exist, it will be created. In this program, we ask the user how many lines he or she wants to append to the file. At lines 18–37, we loop until the user inputs an integer whose value is between 20 and 50, inclusive. We use a *boolean* variable, *goodInput*, to monitor the validity of the value entered by the user. As long as *goodInput* is *false* (line 37), we keep asking the user for a new value. If the user does not enter an integer, we execute the *catch* block and flush the input stream (lines 32–33). If we instead leave the invalid user input in the input stream, we would attempt to read that invalid user input again at line 25 at the next iteration of the loop. If the user enters an integer, we test that the value is between 20 and 50 at line 26. If the input is valid, we change *goodInput* to *true* and exit the loop.

COMMON ERROR TRAP

Opening a file for writing will cause the existing file data to be deleted. If you intend to add new data to a file while maintaining the original contents, open the file for appending.

If the *ItalianFlag.txt* file is the one that was created by Example 11.5, we can use two approaches to check if we did append to the file when we ran this program: We can open the file in Notepad and check that it contains the extra lines. We can also run Example 11.4 again, using the newly created file, and we can verify that the drawing is larger than before.

```
AA123
BWI
SFO
235
239.5
```

```
AA123,BWI,SFO,235,239.5
AA200,BOS,JFK,150,89.3
AA900,LAX,CHI,201,201.8
```

```
Flight[flightNumber=AA123, origin=BWI, destination=SFO,
    numPassengers=235, avgTicketPrice=239.5]
Flight[flightNumber=AA200, origin=BOS, destination=JFK,
    numPassengers=150, avgTicketPrice=89.3]
Flight[flightNumber=AA900, origin=LAX, destination=CHI,
    numPassengers=201, avgTicketPrice=201.8]
```

11.7 Reading Structured Text Files

Sometimes a text file is organized so that each line represents data related to a particular record or object. For instance, an airline company could have data stored in a file where each line represents a flight segment, with the following comma-separated data:

- flight number

- origin airport
- destination airport
- number of passengers
- average ticket price

Such a file could contain the following data:

```
AA123,BWI,SFO,235,239.5

AA200,BOS,JFK,150,89.3

AA900,LAX,CHI,201,201.8

. . .
```

Description

As we read the file, we should **parse** each line, that is, separate the line into the individual pieces of data (flight number, origin airport, etc.), called **tokens.** In this case, the comma is the **delimiter**; that is, a comma separates one token from the next. We will store the tokens from each line into a corresponding *FlightRecord* object.

11.7.1 Parsing a *String* Using *Scanner*

In addition to accepting input from the console and text files, *Scanner* can also be used to parse *Strings*, that is, to separate *Strings* into tokens.

A constructor of the *Scanner* class, which takes a *String* to parse, is shown in **Table 11.7**.

Useful Constructor and Method of the *Scanner* Class for Parsing a *String*

Constructor	**Exceptions Thrown**
`Scanner(`	None

`String source`
`)`

constructs a *Scanner* object that produces tokens from the specified *String*

Method API

Return value	Method name and argument list	

`Scanner`	`useDelimiter(`
	`String pattern`
	`)`
		None

sets this *Scanner* object's delimiters based on the specified *pattern*

Description

Scanner's default delimiters are the white space characters. We can specify different delimiters through the *useDelimiter* method (also shown in Table 11.7), which accepts as its argument a *String* representing a **regular expression**. Regular expressions allow us to specify a pattern against which to match sequences of characters using standard characters as well as **meta-characters**, which have special meanings. Further discussion of regular expressions is outside the scope of this book. For our purposes, we can specify a delimiter consisting of a single or multiple specific characters as a simple *String* argument for the *useDelimiter* method. For example, to parse a *String* like *AA123,BWI,SFO,*235,239.5, we call the *useDelimiter* method and pass a comma as its argument.

Example 11.7 shows how the *Scanner* class can be used in a Java program to parse a *String*.

Example 11.7 Demonstrating How to Parse a *String* with the *Scanner* Class

```
 1 /* Demonstrating how to parse a String with the Scanner class
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6 import java.util.InputMismatchException;
 7 import java.util.NoSuchElementException;
 8
 9 public class UsingScannerToParseAString
10 {
11   public static void main( String [ ] args )
12   {
13     String flightRecord1 = "AA123,BWI,SFO,235,239.5";
14
15     try ( Scanner parse = new Scanner( flightRecord1 ) )
16     {
17       // set the delimiter to a comma
18       parse.useDelimiter( "," );
19
20       System.out.println( parse.next( ) ); // flight number
21       System.out.println( parse.next( ) ); // origin airport
22       System.out.println( parse.next( ) ); // destination airport
23       System.out.println( parse.nextInt( ) ); // number of passengers
24       System.out.println( parse.nextDouble( ) ); // average ticket price
25     }
26
27     catch ( InputMismatchException ime )
28     {
29       System.out.println( "Error in data format" );
30     }
31
32     catch ( NoSuchElementException nse )
33     {
34       System.out.println( "No more tokens" );
35     }
36
37     catch ( IllegalStateException ise )
38     {
39       ise.printStackTrace ( );
40     }
41   }
42 }
```

Description

At line 15, the *Scanner* object *parse* is instantiated using the constructor with a *String* argument, *flightRecord1*, the *String* that we want to tokenize (see line 13). At lines 17–18, we call the *useDelimiter* method to set the delimiter to a comma.

At lines 20–24, we call the *next, nextInt,* and *nextDouble* methods to retrieve the five tokens of *flightRecord1* and process them, echoing them to the console. Since we know the format of the *String flightRecord1*, we can call the appropriate method based on the data type we expect to find at that position in the *String*. Although these methods throw exceptions, these exceptions are unchecked exceptions and thus the *try* and *catch* blocks are not mandatory. The *nextInt, nextDouble*, … methods will throw an *InputMismatchException* if the token retrieved cannot be converted to the expected data type. They will throw a *NoSuchElementException* if there are no more tokens to retrieve. Finally, they will throw an *IllegalStateException* if the *Scanner* object has been closed. We *catch* all these exceptions at lines 27–40, from the most specific to the most general and in the order in which they are most likely to be thrown.

REFERENCE POINT

You can read more about the *Scanner* class at *www.oracle.com/technetwork/java.*

When the program in Example 11.7 runs, it will produce the output shown in **Figure 11.11**.

Figure 11.11 Output from Example 11.7

Description

If we want to process all the tokens as *Strings* with a loop construct, we can call the *hasNext* method to check if there are more tokens to process as in the following:

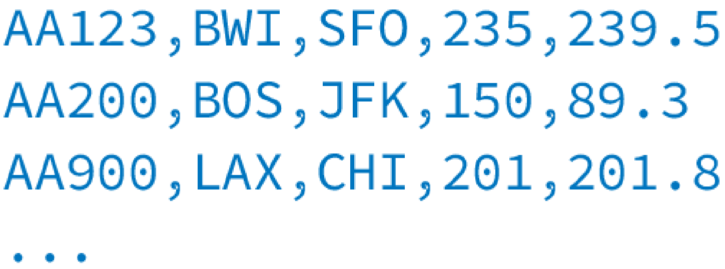
```
while ( parse.hasNext( ) )
  System.out.println( parse.next( ) );
```

Description

Eventually, when all tokens have been retrieved, the *hasNext* method returns *false*, which causes us to exit the *while* loop.

11.7.2 Reading Structured Data Using Scanner

Let’s say that we have a file named *flights.txt* containing many flight records, and we want to read the data into variables. Suppose that each line in the file is in the same format as the *flightRecord1 String* in Section 11.7.1; that is, the file looks like the following:



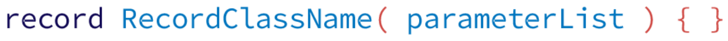
```
AA123,BWI,SFO,235,239.5
AA200,BOS,JFK,150,89.3
AA900,LAX,CHI,201,201.8
...
```

Description

where each line represents a flight segment with the following comma-separated data: flight number, origin airport, destination airport, number of passengers, and average ticket price.

First, we build a class called *Flight*, encapsulating a flight record as reflected by the data in the file. Each line read from the file will be parsed and used to instantiate a *Flight* object. Since we do not know how many lines (i.e., how many flight records) are in the file, we place all the flight records into an *ArrayList* object as opposed to a fixed-length array.

A recent and useful addition to Java is the concept of a ***Record*** class. A *Record* class is a simplified version of a class with instance variables and no functionality. The basic syntax for a Record class is as follows:



```
record RecordClassName( parameterList ) { }
```

Description

This is the equivalent, in a single line, to writing a class with instance variables, an overloaded constructor, accessors, as well as an *equals* and *toString* method. The name of the accessors does not start with get; it is the same as the parameters listed in the parameter list. A *Record* class is very convenient when all we need is to store data without manipulating it. **Example 11.8** shows the *Flight Record* class. The corresponding accessors are automatically generated and are named *flightNumber*, *origin*, *destination*, *numPassengers*, and *avgTicketPrice*. Note that not only are there no mutators generated, but the instance variables are *final*. It is also possible to add static instance variables and methods to a

Record class, as well as to override the accessors.

Example 11.8 The *Flight Record* Class

```java
/* The Flight Record class
   Anderson, Franceschi
*/

record Flight( String flightNumber,
               String origin, String destination,
               int numPassengers, double avgTicketPrice ) { }
```

Description

Example 11.9 shows our client class, which reads the file *flights.txt*, parses each line using *Scanner*, then instantiates a *Flight* object and adds it to the *ArrayList* named *listFlights*.

Example 11.9 Demonstrating How to Read Structured Data from a File

```
/* Reading structured data from a text file
   Anderson, Franceschi
*/

import java.io.File;
import java.io.FileNotFoundException;

import java.util.Scanner;
import java.util.InputMismatchException;
import java.util.ArrayList;

public class ReadFlights
{
   public static void main( String [ ] args )
   {
     // instantiate ArrayList to hold Flight objects
     ArrayList<Flight> listFlights = new ArrayList<Flight>( );

     try
     {
       Scanner file = new Scanner( new File( "flights.txt" ) );

       while ( file.hasNext( ) ) // test for the end of the file
       {
          // read a line
          String stringRead = file.nextLine( );

          // process the line read
          Scanner parse = new Scanner( stringRead );
          parse.useDelimiter( "," );
          String flightNumber = parse.next( );
          String origin = parse.next( );
          String destination = parse.next( );

          try
          {
             int numPassengers = parse.nextInt( );
             double avgTicketPrice = parse.nextDouble( );

             Flight frTemp = new Flight(
                    flightNumber, origin, destination,
                    numPassengers, avgTicketPrice );

             // add Flight obj to listFlights
             listFlights.add( frTemp );
          }

          catch ( InputMismatchException ime )
          {
             System.out.println( "Error in flight record: "
                       + stringRead + "; record ignored" );
          }
       }

       // release resources associated with flights.txt
       file.close( );
     }

     catch ( FileNotFoundException fnfe )
     {
        System.out.println( "Unable to find flights.txt" );
     }

     catch ( Exception ioe )
     {
        ioe.printStackTrace( );
     }

     // print the Flights read
     for ( Flight flight : listFlights )
         System.out.println( flight );
   }
}
```

Description

Lines 5–10 import the classes needed for input, parsing, and exception handling, as well as *ArrayList*. The *Flight Record* class is also used in this program, but is assumed to be in the same folder as the *ReadFlights* class.

In this example, we instantiate two *Scanner* objects: one to read the lines from the file into a *String* and a second *Scanner* object to parse the *String*.

With the first *Scanner* object (instantiated on line 21), we use the *nextLine* method to read one line at a time (line 26). We provide *catch* blocks for both *FileNotFoundException* (lines 59–62) and *Exception* (lines 64–67). The *Scanner* constructor can throw a *FileNotFoundException* and the *File* constructor can throw a *NullPointerException*. Both *FileNotFoundException* and *NullPointerException* are subclasses of *Exception*. Remember that when an exception occurs, the *catch* blocks are scanned, in order, for a match between the *catch* block parameter and the type of exception that occurred. Because the *FileNotFoundException* is a subclass of *Exception*, a *FileNotFoundException* will also match a *catch* block for an *Exception*. Therefore, we need to put the *catch* block for the *FileNotFoundException* before the *catch* block for *Exception*. This way, if the file does not exist, the exception will match the first *catch* block, which handles the *FileNotFoundException*, and we will be able to output a meaningful message for the user. If the *File* constructor throws a *NullPointerException*, which we don’t expect to happen, that will match the *catch* block for *Exception* and we print the stack trace.

As we read each line from the file, we instantiate a second *Scanner* object (line 29) and pass the data we extract as our arguments to the *Flight* constructor to instantiate a *Flight* object (lines 40–42).

The *Flight* object is then added to the *ArrayList listFlights* at lines 44–45. The *ArrayList listFlights* is declared and instantiated at lines 16–17, before the *try* block so that *listFlights* is available for printing the *Flight* objects at lines 69–71, after we finish reading the file. If the *flights.txt* file contains the data shown in **Figure 11.12**, the program will produce the output shown in **Figure 11.13**.

Figure 11.12 Contents of *flights.txt*

Description

Figure 11.13 Output from *ReadFlights.java*

Description

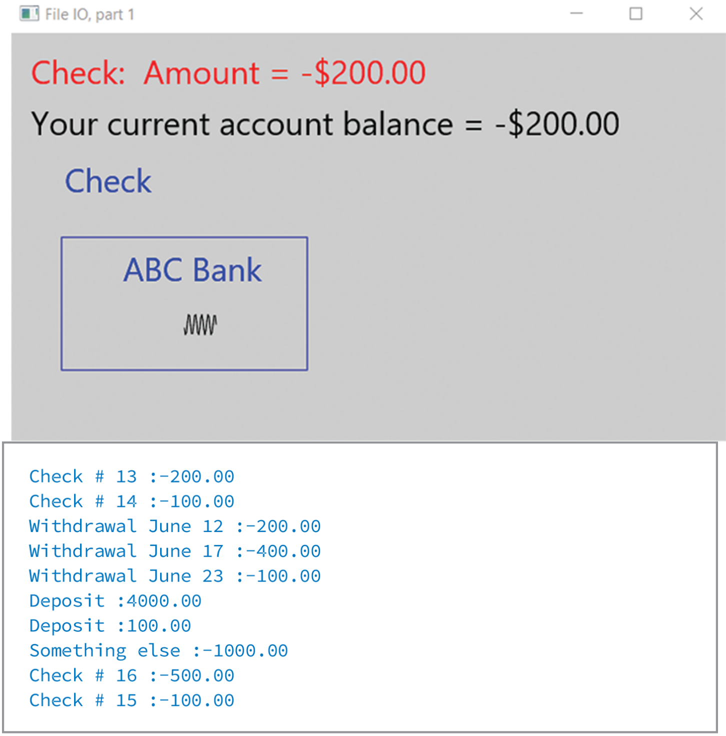
CODE IN ACTION

Within the online resources, you will find a movie with a step-by-step illustration of reading from and writing to a text file. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

with these end-of-chapter questions

11.15.1	Multiple Choice Exercises Questions 1, 2, 4, 5, 6, 7, 8, 9
11.15.2	Reading and Understanding Code Questions 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23
11.15.3	Fill In the Code Questions 24, 25, 26, 28, 29, 30, 31, 32, 33
11.15.4	Identifying Errors in Code Questions 34, 35, 36
11.15.5	Debugging Area Questions 37, 38, 39, 40, 41, 42
11.15.6	Write a Short Program Questions 43, 44, 45, 46, 47, 48, 50, 51



Check: Amount = -$200.00

Your current account balance = -$200.00

```
Check # 13 :-200.00
Check # 14 :-100.00
Withdrawal June 12 :-200.00
Withdrawal June 17 :-400.00
Withdrawal June 23 :-100.00
Deposit :4000.00
Deposit :100.00
Something else :-1000.00
Check # 16 :-500.00
Check # 15 :-100.00
```

11.8 Programming Activity 1: Reading from a Structured Text File

In this activity, you will read from a text file using an end-of-file controlled *while* loop performing this activity:

> Read a text file containing transaction items for a bank account. Loop through all the transaction items and calculate the new balance of the bank account. Assume that we do not know the number of transaction items (i.e., lines) in the file.

The framework will display the current transaction and current balance so that you can check the correctness of your code as the program executes.

For example, **Figure 11.14** demonstrates the animation: We are currently scanning a check for the amount of $200.00. The original balance was $0.00 and the new balance is –$200.00. Ideally, this is not your bank account.

Figure 11.14 Animation Showing a $200 Check and the New Balance

Description

Instructions

In this chapter's folder in the supplied code, you will find the Programming Activity 1 folder. Copy the contents of the folder onto a folder on your disk.

- Open the file *transactions.txt* with a text editor. You will see that each line contains a transaction name and transaction amount separated by a colon, as shown in **Figure 11.15**.
- Note that the transaction amounts are positive or negative. For instance:
    - A check or a withdrawal has a negative amount.
    - A deposit has a positive amount.
    - An unknown transaction has either a positive or negative amount.
- Now open the *AccountingDrawing.java* file. Search for five asterisks (*****) to find the *balanceCheckBook* method where you will add your code. The method header has already been coded for you. Write the code to read all the transactions from the file *transactions.txt*, process each transaction against the account, and calculate the balance after all the transactions in that file have been processed.

Figure 11.15 Contents of the *transactions.txt* File

Description

The code for the *balanceCheckBook* method is shown in **Example 11.10**.

Example 11.10 The *balanceCheckBook* Method in *Accounting Drawing.java*

```
public void balanceCheckBook( )
{
  // ***** Write the body of this method *****
  //
  // Using a while loop, read the file transactions.txt
  // The file transactions.txt contains money
  // transactions between you and your bank
  //
  // You will need to call the method animate inside
  // the body of the loop reading the file contents
  //
  // The animate method takes 3 arguments:
  //    a String, representing the type of transaction
  //    a double, representing the transaction money amount
  //    a double, representing the new checkbook balance
  // So if these 3 variables are:
  //    transactionName, currentAmount, and balance,
  // then the call to animate will be:
  //
  //  animate( transactionName, currentAmount, balance );
  //
  // You should make that call in the body of your while
  // loop, after you have updated the checkbook balance
  //
  //
  // end of student code
  //
}
```

Description

- Begin with a checkbook balance of 0.00.
- To process the transactions, you will need to read one line at a time from the *transactions.txt* file and parse the *String* that you retrieve. You can use the *Scanner* class for this. The delimiter will be a colon. Then process the transaction; you do not need to check the type of transaction. Just add the amount of the transaction to the checkbook balance. Adding a negative transaction amount will decrease the balance, as expected. Be sure to use *try/catch* blocks where appropriate.
- After you have processed each transaction, call the *animate* method. This method belongs to the *AccountingDrawing* class, so you will call *animate* without using an object reference. The API of the *animate* method is the following:

```
public void animate( String currentTransaction,
                     double currentAmount,
                     double currentBalance )
```

Description

- As you can see, the *animate* method takes three arguments: *currentTransaction* is the transaction name ("Deposit," for example), *currentAmount* is the amount of the transaction (-*45.00*, for example), and *currentBalance* is the current balance of the checkbook. Assuming that you have a *String* variable called *transactionName*, a *double* variable called *amount*, and another *double* called *balance*, a call to *animate* will look like the following:

```
animate( transactionName, amount, balance );
```

Description

- When you call *animate*, the window will display the current transaction graphically. It will also display the transaction amount (red if negative, blue if positive), and the current checkbook balance (in black). By adding the previous checkbook balance to the current transaction amount, you will be able to determine if your program is working correctly.
- When you reach the end of the file, print the final balance and write it to a file named *balance.txt*.

To test your code, compile and run the *AccountingApplication.java* application file.

If you have time …

- Modify the file *transactions.txt* by deleting or adding transactions manually with a text editor. Run the program again and check that your code still gives the correct result.
- Using a text editor, modify the file *transactions.txt* by entering a positive amount to all transactions. Change your *balanceCheckBook* method so that it determines which

transactions are positive and which are negative. Run the program again and check that your code still gives the correct result.

Troubleshooting

If your method implementation does not animate or animates incorrectly, check these items:

- Verify that you coded the call to *animate* at the proper time.
- Verify that you coded the condition for exiting the loop correctly.
- Verify that you coded the body of the loop correctly.

DISCUSSION QUESTIONS

1. What exceptions can occur during this program?
2. Explain why we use the *Scanner* class.

```
Paris
Baltimore
Seattle
San Francisco
Boston
Los Angeles
Madrid
Barcelona
Buenos Aires
```

```
1100.50
-100.48
200.39
150.00
```

```
Number of cities starting with B: 4

Total expenses are $1,450.89
```

11.9 Streams

Streams were introduced with Java 8 to support aggregate sequential and parallel operations on a collection of elements. The *Stream* interface (*java.util.stream* package) specifies functionality to perform those operations on a collection of objects. It uses generics, so we need to specify the type of object that a *Stream* will be working on. The *java.util.stream* package includes more specialized stream interfaces such as *IntStream*, *DoubleStream*, or *LongStream*. *Streams* cannot directly access individual elements and thus are designed for performing computations on all or a subset of their elements.

A *Stream* computation goes through a **stream pipeline**. A stream pipeline includes a source of elements that is first converted into a *Stream*; then zero or more operations are performed on those elements, for example filtering or selecting, resulting in another *Stream*. A final operation, for example counting or summing, is then performed, producing a result. **Table 11.8** lists some of the methods of the *Stream* interface.

Selected Methods of the *Stream* Interface

Return Value	Method and Description
`long`	`count( )` returns the number of elements in this *Stream*
`Stream`	`filter( Predicate<? super T> predicate )` returns a *Stream* made of the elements of this *Stream* that satisfy *predicate*
`Optional`	`min( Comparator<? super T> comparator )` returns the minimum element of this *Stream* using the comparator
`Optional`	`max( Comparator<? super T> comparator )` returns the maximum element of this *Stream* using the comparator
`DoubleStream`	`mapToDouble( ToDoubleFunction <? super T > mapper )` returns a *DoubleStream* made up of the elements resulting from applying *mapper* to the *Stream* elements; *mapToInt* and *mapToLong* are similar methods

Description

Let's take a closer look at how we can use these methods, for example the *filter* method. If we want to process only the data in the *Stream* that meets some condition, we can use the *Predicate* interface in the *java.util.function* package. The *Predicate* interface uses generics and allows us to test if the data meets a condition. The *filter* method accepts a parameter of type *Predicate<? super T>.* That means the data type of the *Stream* being tested can be of class *T* or a superclass of *T*. Thus, if *cityPredicate* tests if a *String* meets some condition and *cityStream* is an existing *Stream* of type *String*, we can call the *filter* method as follows:

```
cityStream = cityStream.filter( cityPredicate );
```

Description

The *Predicate* interface has only one *abstract* method, *test*. **Table 11.9** shows the *test* method as well as the *abstract* methods of the *DoublePredicate, Comparator,* and *ToDoubleFunction* functional interfaces. Note that except for *DoublePredicate*, all these functional interfaces use generics. If we want *cityPredicate* to filter *Strings* that start with the letter *B*, we can implement the *test* method of the *Predicate* interface as shown in **Example 11.11**.

Methods of the *Predicate, DoublePredicate, Comparator*, and *ToDoubleFunction* Interfaces

Interface	**Abstract Method**
`Predicate`	`boolean test( T t )` tests *t* and returns *true* or *false*
`DoublePredicate`	`boolean test( double d )` tests *d* and returns *true* or *false*
`Comparator`	`int compare( T t1, T t2 )` compares *t1* and *t2* for order
`ToDoubleFunction`	`double applyAsDouble( T t )` converts *t* to a *double* and returns that value

Description

Example 11.11 Implementing the *Predicate* Interface

```
/*  Predicate testing whether a String starts with B
    Anderson, Franceschi
*/
import java.util.function.*;

public class StartWithBPredicate<T> implements Predicate<T>
{
  public boolean test( T t )
  {
    if ( !( t instanceof String ) )
      return false;
    else
    {
      String s = ( String ) t;
      return s.charAt( 0 ) == 'B';
    }
  }
}
```

Description

Below is how we can use the *StartWithBPredicate* test method to filter our *Stream*:

```
StartWithBPredicate<String> cityPredicate
    = new StartWithBPredicate<String>( );
cityStream = cityStream.filter( cityPredicate );
```

Description

Because the *Predicate* interface contains only one *abstract* method that must be implemented, it is a **functional interface**. Thus, we can use a **lambda expression** to replace the code above, as well as the code in Example 11.11, in a single statement as follows:

```
cityStream = cityStream.filter( city -> city.charAt( 0 ) == 'B' );
```

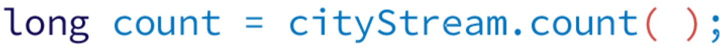
Description

Lambda expressions, introduced in Java version 8, can be used only with functional interfaces, which are interfaces that have only one *abstract* method to implement. A lambda expression contains the following elements:

- A comma-separated list of parameters enclosed in parentheses

- The data types of the parameters may be omitted. The parentheses may also be omitted if there is only one parameter (in the above example, *city* is the parameter).
- The arrow token, ->
- A method body, which can be a single expression or a block enclosed in curly braces
- If the body of the method consists of a single expression, the JVM evaluates the expression and returns its value (in the above example, the JVM returns the value of the expression `city.charAt( 0 ) == 'B'`). As an alternative, we can use a *return* statement, but that requires curly braces.

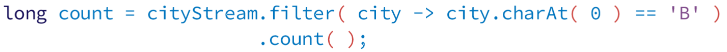
After we have filtered the *Stream*, we may want to perform a final operation, such as counting. In this example, we count how many cities in the *cities.txt* file start with the letter *B*. After filtering the *Stream*, we call the *count* method, which returns a result, as follows:



```
long count = cityStream.count( );
```

Description

Because the *filter* method returns a *Stream* reference, we can chain the method calls as follows:



```
long count = cityStream.filter( city -> city.charAt( 0 ) == 'B' )
                       .count( );
```

Description

COMMON ERROR TRAP

Do not confuse the *Path* interface with the *Paths* class. They work together but are different.

Before processing a *Stream*, we need to create one. **Table 11.10** shows some classes and methods that can be used for that purpose. The *stream* method of the *ArrayList* class, inherited from the *Collection* interface, returns a *Stream* consisting of the elements in the *ArrayList*. To create a *Stream* from the contents of a file, we can call the *static lines* method of the *Files* class and pass a *Path* argument for that file. The *Path* interface and the *Paths* and *Files* classes are all in the *java.nio.file* package. A *Path* can be obtained for a file by calling the *static get* method of the *Paths* class. As shown, the *get* method accepts a variable number of arguments for specifying folders in the path to the filename; if the file is located in the current folder, we just pass the name of the file. We can combine the two method calls in one statement as follows:

Useful Classes and Methods for Creating a *Stream*

Class	**Method**	**Exceptions Thrown**
`Paths`	`static Path get( String first, String... more )` returns a *Path* constructed with the *String* parameters	`InvalidPathException`
`Files`	`static Stream<String> lines( Path p )` reads all the lines from *p* and returns them as a *Stream*	`IOException, SecurityException`
`ArrayList`	`Stream stream( )` returns a *Stream* consisting of the elements in this *ArrayList*	

Description

```
Stream<String> cityStream = Files.lines( Paths.get( "cities.txt" ) );
```

Description

Example 11.12 shows how we can use *Streams* to perform some aggregate operations on the contents of two files, *cities.txt* and *expenses.txt*, shown in **Figures 11.16** and **11.17**.

Figure 11.16 The *cities.txt* File

Description

Figure 11.17 The *expenses.txt* File

Description

Example 11.12 Processing *Streams*

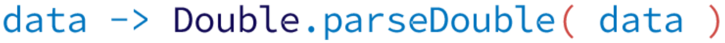
```
 1 /*  Reading file data into a Stream and processing it
 2 *   Anderson, Franceschi
 3 */
 4
 5 import java.io.IOException;
 6 import java.nio.file.*;
 7 import java.text.NumberFormat;
 8 import java.util.stream.Stream;
 9
10 public class FilterAndProcessStream
11 {
12   public static void main( String [ ] args  )
13   {
14    NumberFormat money = NumberFormat.getCurrencyInstance( );
15    String cityFile = "cities.txt";
16    String expenseFile = "expenses.txt";
17    try
18    {
19      // build a Stream using cities.txt
20      Stream<String> cityStream =
21        Files.lines( Paths.get( cityFile ) );
22      // select cities starting with a B and count them
23      long count = cityStream.filter( city -> city.charAt( 0 ) == 'B' )
24                             .count( );
25      System.out.println( "Number of cities starting with B: " + count );
26
27      // build a Stream using expenses.txt
28      Stream<String> numberStream =
29        Files.lines( Paths.get( expenseFile ) );
30      // select positive costs and add them up
31      double sum =
32        numberStream.mapToDouble( data -> Double.parseDouble( data ) )
33                    .filter( cost -> cost > 0.0 )
34                    .sum( );
35      System.out.println( "\nTotal expenses are " + money.format( sum ) );
36    }
37
38    catch ( InvalidPathException ipe )
39    {
40      System.out.println( ipe.getMessage( ) );
41    }
42
43    catch ( IOException ioe )
44    {
45      System.out.println( "Could not find file: " + ioe.getMessage( ) );
46    }
47
48    catch ( SecurityException se )
49    {
50      System.out.println( se.getMessage( ) );
51    }
52   }
53 }
```

Description

We count the number of cities that begin with *B* in *cities.txt* and calculate the total of the values that are greater than 0.0 in *expenses.txt*.

At lines 19–21, we build a *Stream* using the contents of the *cities.txt* file. At lines 22–24, we compute how many cities start with the letter *B*. We output that count at line 25.

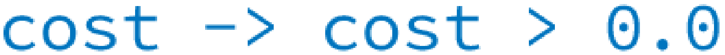
At lines 27–29, we build a *Stream* using the contents of the *expenses.txt* file. At lines 30–34, we select the positive *doubles* from that *Stream* and calculate a sum. We output that total using a currency format at line 35. At line 32, we call the *mapToDouble* method in order to convert the *Stream* of *Strings* to a *DoubleStream*. Since *ToDoubleFunction* is a functional interface, we use the lambda expression



```
data -> Double.parseDouble( data )
```

Description

as the argument of *mapToDouble*. We then call the *filter* method of the *DoubleStream* interface at line 33, which accepts a *DoublePredicate* parameter. Our predicate test determines whether the cost is greater than 0.0. Since *DoublePredicate* is also a functional interface, we use the lambda expression



```
cost -> cost > 0.0
```

Description

as the argument of *filter*. At line 34, we call the *sum* method with the filtered *DoubleStream* in order to sum all its elements. Both the *filter* and *sum* methods are shown in **Table 11.11**.

The *filter* and *sum* Methods of the *DoubleStream* Interface

Return Value	Method and Description
`DoubleStream`	`filter( DoublePredicate predicate )` returns a *DoubleStream* made of the elements of this *DoubleStream* that satisfy *predicate*
`double`	`sum( )`

returns the sum of the elements in this *DoubleStream*

Description

Figure 11.18 shows the output of Example 11.12.

Figure 11.18 Output of Example 11.12

Description

```
city1 is New York

state is FL
```

```
[{"title":"After Dark in Central Park","year":1900,"cast":[],
"genres":[]},{"title":"Boarding School Girls' Pajama Parade",
"year":1900,"cast":[],"genres":[]},{"title":"Buffalo Bill's
Wild West Parade","year":1900,"cast":[],"genres":[]},
...
{"title":"Holmes and Watson","year":2018,"cast":["Will
Ferrell","John C. Reilly","Rebecca Hall","Ralph Fiennes","Rob
Brydon","Kelly Macdonald","Lauren Lapkus","Hugh Laurie"],
"genres":["Action","Mystery","Comedy"]},{"title":"On the
Basis of Sex","year":2018,"cast":["Felicity Jones","Armie
Hammer","Justin Theroux","Jack Reynor","Cailee Spaeny",
"Sam Waterston","Kathy Bates"],"genres":["Biography",
"Drama"]},{"title":"Destroyer","year":2018,"cast":["Nicole
Kidman","Tatiana Maslany","Sebastian Stan","Toby Kebbell",
"Scoot McNairy"],"genres":["Crime","Thriller"]}]
```

```
There are 28795 movies
Please enter a year > 2000
There were 213 movies in 2000
```

11.10 Reading Formatted Open Data from a Remote Location

More and more organizations open their data to the general public, and to programmers in particular. In this section, we show how to access data located on a remote server, parse the data, convert the data to a *Stream*, and process the *Stream*.

11.10.1 Accessing Remote Data

There are many data sets that are open to the public. For this example, we have chosen to retrieve and process data on American movies made since 1900. The data is located at the following Uniform Record Locator (URL):

```
https://raw.githubusercontent.com/prust/wikipedia-movie-data/master/movies.json
```

Description

In order to access the data, we use the *URL* and *Scanner* classes. Some of their constructors and methods are listed in **Table 11.12**. The code sequence below shows how we can access and read the data. The *String resource* stores the name of the *URL* where the data source we want to process is located. We first create and instantiate a *URL* object using the URL of the resource that we want to access. If the resource's URL does not have proper syntax, the *URL* constructor throws a *URLMalformedException*. With the *URL* reference, we then call the *openStream* method, which returns an *InputStream* reference and will throw an *IOException* if there is no resource at that URL. We catch both exceptions. We now can read data from the remote resource with the *Scanner* named *dataSource*.

Useful Classes, Constructors, and Methods for Accessing Remote Data

Class	Constructor or Method	Exceptions Thrown
`URL`	`URL( String url )` constructs a URL object	`MalformedURLExceptio` will be thrown if *url* is not a properly formed URL
`URL`	`InputStream openStream( )` creates and returns an *InputStream* for this URL	`IOException`
`Scanner`	`Scanner( InputStream is )` constructs a *Scanner* object that can read from the *InputStream is*	

Description

```
 String GITHUB = "https://raw.githubusercontent.com/";
 String resource = GITHUB + "prust/wikipedia-movie-data/master/movies.json";
 try
 {
   URL url = new URL( resource );
  InputStream is = url.openStream( );
  Scanner dataSource = new Scanner( is );
   ..
}
catch ( MalformedURLException murle )
{
   ..
}
catch ( IOException ioe )
{
   ..
}
```

Description

11.10.2 JSON Formatting and Parsing

Many open data files are formatted in **XML** or **JSON**. XML stands for eXtended Markup Language. It is similar to HTML except that tags are user-defined instead of predefined. JSON stands for JavaScript Object Notation and is based on JavaScript syntax. JavaScript is an interpreted language that typically runs inside a browser. The JSON format is often the format of choice for transferring data over the Internet between a client and a server. The file that we access in **Example 11.13** is a JSON-formatted file. A JSON string includes two data - structures:

Example 11.13 Parsing a JSON

```
/*  JSON Parsing
*   Anderson, Franceschi
*/

import org.json.*;

public class JsonParsing
{
  public static void main( String [] args )
  {
    try
    {
      String json1 = "[Paris, New York, Rio]";
      JSONArray jsonArr = new JSONArray( json1 );
      String city1 = jsonArr.getString( 1 );
      System.out.println( "city1 is " + city1 );

      String json2 =
        "{'city':'Miami', 'state':'FL', 'country':'USA'}";
      JSONObject jsonObj = new JSONObject( json2 );
      String state = jsonObj.getString( "state" );
      System.out.println( "state is " + state );
    }
    catch( JSONException je )
    {
      System.out.println( "exception: " + je.toString( ) );
    }
  }
}
```

Description

- A JSON array
- A JSON object, representing a mapping of key/value pairs

Those two data structures can be nested. A JSON array is a list of comma-separated values enclosed in square brackets. A JSON object is enclosed in curly braces and includes a comma-separated list of key/value pairs. A colon separates a key from its associated value. Keys are strings enclosed in quotes. Values can be strings, numbers, true, false, null, an object, or an array.

Here are some examples of valid JSON strings:

```
{ "email":"jane45@gmail.com", "age": 21 }
{ "states": { "MD":"Maryland", "NY":"New York", "CA":"California" } }
[ "New York", "London", "Buenos Aires"]
{ "countries": ["USA", "China", "Brazil"] }
```

Description

The first JSON string is a JSON object that contains two key/value pairs.

The second JSON string is a JSON object that contains one key/value pair. The key states maps to a value that is a JSON object itself, containing three key/value pairs.

The third JSON string is a JSON array that contains three values.

The fourth JSON string is a JSON object that contains one key/value pair. The key countries maps to a JSON array that contains three values.

In order to parse a *JSON* string, we can use the *JSONObject* and *JSONArray* classes of the *org.json* package. Some of their constructors and methods are listed in **Table 11.13**. The *org.json* package does not come as part of the standard JDK distribution and needs to be downloaded. After you download (and unpack) the *org.json* package, either place it in the folder where the example is located, or, preferably, update the CLASSPATH environment variable to include the folder where the package is located.

Useful Classes, Constructors, and Methods for Parsing a JSON *String*

Class	Constructor or Method	Exceptions Thrown
`JSONObject`	`JSONObject( String json )` constructs a JSONObject	`JSONException` will be thrown if *json* cannot be parsed into a *JSONObject*
	`dataType getDataType( String key )` returns the value for that *key*. The data type can be a primitive data type or *String*	`JSONException` will be thrown if *key* is not found or the value for *key* is not compatible with the expected return type

`JSONArray`	`JSONArray( String json )` constructs a JSONArray	`JSONException` will be thrown if *json* cannot be parsed into a *JSONArray*
	`int length( )` returns the number of values in this *JSONArray*	
	`JSONObject getJSONObject( int index )` returns the *JSONObject* at *index*	`JSONException` will be thrown if *index* is invalid or there is no *JSONObject* at that *index*

Description

Example 11.13 shows how to use the *JSONArray* and *JSONObject* classes to parse two simple JSON strings. To convert a JSON *String* that contains array-like data, we construct a *JSONArray* object from the JSON *String* (lines 13–14). Since we want to retrieve a *String* value, we call the *getString* method at line 15, passing an integer argument representing the array index of the value we want to retrieve. To convert a JSON *String* that contains object-like data, we construct a *JSONObject* object from the JSON *String* (lines 18–20). Again, we want to retrieve a *String* value, and we call the *getString* method at line 15, passing a *String* argument representing the key for the value we want to retrieve.

Figure 11.19 shows the output of Example 11.13.

Figure 11.19 Output of Example 11.13

Description

Figure 11.20 shows sample contents of the data at the URL discussed previously. The JSON *String* from that URL contains an array of movie objects. All the objects have the same format and contain four key/value pairs. The keys are title, year, cast, and genres; the values for cast and genres are arrays themselves. The last movie object in the array is:

```
{"title":"Destroyer","year":2018,"cast":["Nicole Kidman",
"Tatiana Maslany","Sebastian Stan","Toby Kebbell","Scoot
McNairy"],"genres":["Crime","Thriller"]}
```

Description

Figure 11.20 Sample JSON Content Listing American Movies Data

Description

If we have read the entire data set into a *String* named *json*, we can use the following code sequence to retrieve the title and the year of the movie at index *i* within the array of movies:

```
// json is a String that contains the data in Figure 11.20
JSONArray jsonArray = new JSONArray( json );
// get the JSONObject at index i
JSONObject jsonMovie = jsonArray.getJSONObject( i );
// retrieve the title of the current movie
String title = jsonMovie.getString( "title" );
// retrieve the year of the current movie
int year = jsonMovie.getInt( "year" );
```

Description

First, we create and instantiate a *JSONArray* using the *String json*. With it, we call the *getJSONObject* method, passing the valid array index. The resulting *JSONObject*, *jsonMovie*, is a *JSONObject* with four key/value pairs, storing the data for one movie. We retrieve the title of that movie by calling the *getString* method and passing *title*. We retrieve the year of that movie by calling the *getInt* method and passing *year*.

11.10.3 Reading, Parsing, Streaming, and Processing Remote Data

Example 11.14 shows the *Movie Record* class. We just use that class to store data so it makes sense to use a *Record* class. To keep things simple, we only store the title and the year of each movie.

Example 11.14 The *Movie Record* Class

```
/* The Movie Record class
   Anderson, Franceschi
*/

record Movie( String title, int year ) { }
```

Description

Example 11.15 shows how we can read remote data, then parse it and process it. At lines 14–16, we declare a *String* storing the URL resource that we want to access. At lines 20–22, we create a *URL* for that *String*, an *InputStream* for the *URL*, and a *Scanner* ready to read from that *InputStream*. At lines 23–29, we read all the lines from the data source and accumulate them into a *String* named *json*. At lines 31–33, we convert *json* to a *JSONArray* and retrieve and output the number of elements (i.e., movies) in that *JSONArray*. Because the *JSONArray* stores all the movies, calling the *length* method of the *JSONArray* class at line 32 returns the number of movies in the data set. There are 28,795 movies.

Example 11.15 Reading, Parsing, and Processing Remote Data

```java
/* A class to analyze data from movies
   Anderson, Franceschi
*/

import java.io.*;
import java.net.*;
import java.util.*;
import org.json.*;

public class Movies
{
  public static void main( String args [ ] )
  {
    String GITHUB = "https://raw.githubusercontent.com/";
    String fileName = GITHUB +
      "prust/wikipedia-movie-data/master/movies.json";

    try
    {
      URL url = new URL( fileName );
      InputStream is = url.openStream( );
      Scanner file = new Scanner( is );
      String json = "";
      String s = null;
      while ( file.hasNext( ) )
      {
        s = file.nextLine( );
        json += s;
      }

      JSONArray jsonArray = new JSONArray( json );
      System.out.println( "There are " + jsonArray.length( )
                                       + " movies" );

      // Build ArrayList of MovieRecord objects
      ArrayList<Movie> movies = new ArrayList<Movie>( );

      for ( int i = 0; i < jsonArray.length( ); i++ )
      {
        JSONObject jsonMovie = jsonArray.getJSONObject( i );
        String title = jsonMovie.getString( "title" );
        int year = jsonMovie.getInt( "year" );
        if ( title.equals( "" ) )
          System.out.println( "Discarding (no title)" );
        else
          movies.add( new Movie( title, year ) );
      }

      // Ask user to enter a year
      Scanner scan = new Scanner( System.in );
      System.out.print( "Please enter a year > " );
      int year = scan.nextInt( );

      // Retrieve number of movies for that year
      long count = movies.stream( )
                         .filter( movie -> movie.year( ) == year )
                         .count( );
      System.out.println( "There were " + count
                          + " movies in " + year );
    }
    catch( MalformedURLException murle )
    {
      System.out.println( murle.getMessage( ) );
    }
    catch( IOException ioe )
    {
      System.out.println( ioe.getMessage( ) );
    }
    catch( JSONException jsone )
    {
      System.out.println( jsone.getMessage( ) );
    }
  }
}
```

Description

At lines 38–47, we loop through the *JSONArray jsonArray* collecting the title and year data into the *movies ArrayList* of *Movies*, which we declare and instantiate at line 36.

Sometimes open data contain erroneous or invalid data, and we want to process only valid data. Thus, we test if the movie title is the empty *String* at line 43 and add the corresponding *Movie* object to the *ArrayList* at line 46, only if the title is not empty. This is called **cleaning the data**.

After we have collected the movies, we ask the user to enter a year. At lines 54–59, we compute and output the number of movies for that year. At lines 55–57, we convert the *ArrayList* to a *Stream* and compute the number of movies for the year entered by the user.

Figure 11.21 shows the output of Example 11.15 when the user enters the year 2000.

Figure 11.21 Output of Example 11.15

Description

```
End of the file reached
Closing file
There are 2 flights from BWI
Average number of passengers: 200.0
```

11.11 Reading and Writing Objects to a File

Throughout this text, we have emphasized the benefits of object-oriented programming. Just as we can write text and primitive data types to a file and subsequently read them from the file, we can also write objects to a file and subsequently read them as objects. This is convenient for two reasons:

- We can write these objects directly to a file without having to convert the objects to primitive data types or *Strings*.
- We can read the objects directly from a file, without having to read *Strings* and convert these *Strings* to primitive data types in order to instantiate objects.

To read objects from a file, the contents of the file must have been written as objects. So our first order of business should be to learn how to write objects to a file.

11.11.1 Writing Objects to Files

The *ObjectOutputStream* class, coupled with the *FileOutputStream* class, provides the functionality to write objects to a file. The *FileOutputStream* class, a subclass of *OutputStream,* is used to write raw data, rather than characters, to a file.

The *ObjectOutputStream* class, also a subclass of *OutputStream*, provides a convenient way to write objects to a file. Its *writeObject* method takes one argument—the object to be written.

The classes, constructors, and methods we will use are shown in **Table 11.14**.

Useful Classes, Constructors, and Methods for Writing Objects to a File

Classes, Constructors, and Methods for Writing Objects to a File

Class	**Constructor**	**Exceptions thrown**
`FileOutputStream`	`FileOutputStream( String filename, boolean mode )` constructs a *FileOutputStream* object from a *String* representing the name of a file; if *mode* is *false*, we will write to the file; if *mode* is *true,* we will append to the file	`FileNotFoundExceptio`
`ObjectOutputStream`	`ObjectOutputStream( OutputStream out )` creates an *ObjectOutputStream* that writes to the *OutputStream out*	`IOException`
	Method API	

`ObjectOutputStream`	`void writeObject( Object o )`
	writes the object argument to a file. That object must be an instance of a class that implements the *Serializable* interface. Otherwise, a run-time exception will be generated
		`IOException,`
		`NotSerializableExcept`
		`InvalidClassExceptio`

Description

We will use the *Flight Record* class developed earlier in the chapter. However, in order for an object to be written to a file (and later to be read using the *ObjectInputStream* class), that object must implement the *Serializable* interface. When an object implements the *Serializable* interface, its state can be converted to a byte stream to be written to a file, such that this byte stream can be converted back into a copy of the object when read from the file. Therefore, our modified *Flight2 Record* class will implement *Serializable*, which is in the *java.io* package.

The *Serializable* interface has no methods to implement. As a result, the only things we have to worry about when writing a class implementing *Serializable* are the following:

- the *import* statement
- the class header showing the class *implements Serializable*

Example 11.16 shows the *Flight2 Record* class. This class is identical to the *FlightRecord* class except that it *imports Serializable* (line 5), and *implements* the *Serializable* interface (line 10).

Example 11.16 The *FlightRecord2* Class

```
 1 /* The Flight2 Record class
 2    Anderson, Franceschi
 3 */
 4
 5 import java.io.Serializable;
 6
 7 record Flight2( String flightNumber,
 8                 String origin, String destination,
 9                 int numPassengers, double avgTicketPrice )
10                 implements Serializable { }
```

Description

Example 11.17 shows how the *FileOutputStream* and *ObjectOutputStream* classes can be used in a Java program to write *Flight2* objects to a file named *objects*.

Example 11.17 Writing Objects to a File

```
 1 /* Demonstrating how to write objects to a file
 2    Anderson, Franceschi
 3 */
 4
 5 import java.io.FileOutputStream;
 6 import java.io.ObjectOutputStream;
 7 import java.io.FileNotFoundException;
 8 import java.io.IOException;
 9
10 public class WritingObjects
11 {
12   public static void main( String [ ] args )
13   {
14    // instantiate the objects
15    Flight2 fr1 = new Flight2( "AA31", "BWI", "SFO", 200, 235.9 );
16    Flight2 fr2 = new Flight2( "CO25", "LAX", "JFK", 225, 419.9 );
17    Flight2 fr3 = new Flight2( "US57", "BWI", "DEN", 175, 179.5 );
18
19    try
20    {
21     FileOutputStream fos = new FileOutputStream
22                                   ( "objects", false );
23              // false means we will write to objects
24
25     ObjectOutputStream oos = new ObjectOutputStream( fos );
26
27     // write the objects to the file
28     oos.writeObject( fr1 );
29     oos.writeObject( fr2 );
30     oos.writeObject( fr3 );
31
32     // release resources associated with the objects file
33     oos.close( );
34    }
35
36    catch ( FileNotFoundException fnfe )
37    {
38      System.out.println( "Unable to write to objects" );
39    }
40
41    catch ( IOException ioe )
42    {
43      ioe.printStackTrace( );
44    }
45  }
46 }
```

Description

Lines 14–17 declare and instantiate three *Flight2* objects that we will write to the *objects* file.

Lines 21 and 22 instantiate a *FileOutputStream* object for writing to the *objects* file, then line 25 instantiates an *ObjectOutputStream* object, which we will use to write the *Flight2* objects to the file.

At lines 27–30, using the *ObjectOutputStream* object *oos*, we call the *writeObject* method, passing the three *Flight2* objects we instantiated. The *writeObject* method takes a *Serializable* object as its parameter, here a *Flight2* object, and writes it to the file in such a way that the stream of bytes can be read using the *readObject* method from the *ObjectInputStream* class. Both the *ObjectOutputStream* constructor and the *writeObject* method can throw an *IOException*, which will be caught at line 41.

After this program is executed, the *objects* file will contain a representation of the three *Flight2* objects.

One more note about writing objects to files: A file containing objects can be quite large. Not only does the object data get written to the file, but also the name of the class, a description of each data field, and other information needed to reconstruct the objects when the file is subsequently read.

The *writeObject* method, however, does not write any *static* class variables to the file. Thus, we may consider declaring any constants as *static*, if appropriate.

The *writeObject* method does not write to the file any instance variable that is declared to be *transient*. Thus, we can save space in the file by declaring an instance variable as *transient*. An instance variable is a good candidate to be declared *transient* if we can easily reproduce its value, or if the variable has a value of 0 at the time the file is created. For example, suppose we have an instance variable whose value can easily be recalculated, we can declare it as *transient*; then, that instance variable will not be written to the object file.

SOFTWARE ENGINEERING TIP

To save disk space when writing to an object file, declare the class data as *static* or *transient* where appropriate.

We declare an instance variable as *transient* by inserting the keyword *transient* between the access modifier and the data type of the instance variable, as in the following syntax:

```
accessModifier transient dataType instanceVariableName
```

Description

11.11.2 Reading Objects from Files

Reading objects from a file somewhat parallels writing objects to a file.

The class *ObjectInputStream*, a subclass of *InputStream*, coupled with *FileInputStream*, provides the functionality we need. The *FileInputStream* class is used to read streams of binary data from a file.

ObjectInputStream is designed to read objects from a file. The *readObject* method, which does not take any arguments, reads the next object from the file and returns it. Because the *readObject* method returns a generic *Object*, we must type cast the returned object to the appropriate class. When the end of the file is reached, the *readObject* method throws an *EOFException*. This is in contrast to the *Scanner* class, which provides the *hasNext* method to test whether the end of the file has been reached.

The classes, constructors, and methods discussed previously are shown in **Table 11.15**.

Useful Classes, Constructors, and Methods for Reading Objects from a File

Classes, Constructors, and Methods for Reading Objects from a File		
Class	**Constructors**	**Exceptions thrown**
`FileInputStream`	`FileInputStream( String filename )` constructs a *FileInputStream* object from a *String* representing the name of a file	`FileNotFoundException`
`ObjectInputStream`	`ObjectInputStream( InputStream in )`	`IOException`

constructs an *ObjectInputStream* from the *InputStream in*

Method API

`ObjectInputStream`	`Object readObject( )`
	reads the next object and returns it. The object must be an instance of a class that implements the *Serializable* interface.When the end of the file is reached, an *EOFException* is thrown
		`IOException,`
		`ClassNotFoundExceptio`
		`EOFException`

Description

Example 11.18 shows how these *FileInputStream* and *ObjectInputStream* classes can be used in a Java program to read objects from a file. We assume that the file *objects* contains *Flight2* objects, as written in the previous section.

Example 11.18 Reading, Filtering, and Processing Objects

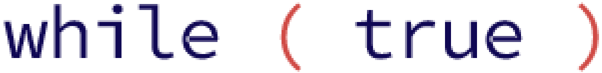
```java
/* Demonstrating how to read objects from a file
   Anderson, Franceschi
*/

import java.io.*;
import java.util.ArrayList;

public class ReadingObjectsIntoStream
{
  public static void main( String [ ] args )
  {
    ArrayList<Flight2> flights = new ArrayList<Flight2>( );
    try
    {
      FileInputStream fis = new FileInputStream ( "objects" );
      ObjectInputStream ois = new ObjectInputStream ( fis );

      try
      {
        while ( true )
        {
          // read object, type cast returned object to Flight2
          Flight2 temp = ( Flight2 ) ois.readObject( );

          // add the Flight2 object read to flights
          flights.add( temp );
        }
      } // end inner try block

      catch ( EOFException eofe )
      {
        System.out.println( "End of the file reached" );
      }

      catch ( ClassNotFoundException cnfe )
      {
        System.out.println( cnfe.getMessage( ) );
      }

      finally
      {
        System.out.println( "Closing file" );
        ois.close( );
      }
    } // end outer try block

    catch ( FileNotFoundException fnfe )
    {
      System.out.println( "Unable to find objects" );
    }

    catch ( IOException ioe )
    {
      ioe.printStackTrace( );
    }

    // calculate number of flights originating from BWI
    long count =
      flights.stream( )
             .filter( flight -> flight.origin( ).equals( "BWI" ) )
             .count( );
    System.out.println( "There are " + count + " flights from BWI" );

    // calculate average number of passengers on all flights
    double avgNumPassengers =
      flights.stream( )
             .mapToInt( Flight2::numPassengers )
             .average( )
             .getAsDouble( );
    System.out.println( "Average number of passengers: "
                      + avgNumPassengers );
  }
}
```

Description

Lines 5–6 import the needed classes from the *java.io* and *java.util* package. The *ClassNotFoundException* class is part of the *java.lang* package and does not need to be imported.

Line 15 associates a *FileInputStream* object with the *objects* file, and line 16 instantiates an *ObjectInputStream* object for reading the objects from the file.

The *while* loop, from lines 20 to 27, reads and adds each object in the file to the *ArrayList flights*, declared and instantiated at line 12. We continue reading until the *readObject* method throws an *EOFException*, which transfers control to the *catch* block (lines 30–33). Thus, our condition for the *while* loop is



```
while ( true )
```

In that *catch* block, we print a message that the end of the file was detected. Given this *while* loop construction, we do not need a priming read. Inside the *while* loop, we read an object, then print it. When the end of the file is detected, the statement adding the current object to *flights* (line 26) will not be executed.

On line 23, we read an object from the file and assign it to the *Flight2* object reference *temp*. Because the *readObject* method returns an *Object*, we need to type cast the return value to a *Flight2* object. The *readObject* method can also throw a *ClassNotFoundException* or an *IOException*, which will be caught at lines 35 or 52, respectively.

Because an *EOFException* will occur when the end of the file is reached, the *EOFException catch* block will always execute in a normal program run. Thus, any code following the *while* loop in the *try* block will not execute. To close the *objects* file, we use nested *try/catch* blocks. The inner *try* block (lines 18–28) encloses the *while* loop; its associated *catch* blocks handle the *EOFException* and *ClassNotFoundException*. The outer *try* block (lines 13–45) encloses the instantiations of the *FileInputStream* and *ObjectInputStream* objects, the inner *try* block, and the *finally* block where we close the file (line 43). We can close the file in the *finally* block because the *ois* object reference, declared in the outer *try* block, is visible (that is, in scope) inside the *finally* block.

The *catch* blocks following the outer *try* block handle any *FileNotFoundException* and any other *IOExceptions* that occur in the inner or outer *try* blocks.

It is important to place the *catch* clause with the *EOFException* ahead of the *catch* clause with the *IOException*; otherwise, the *EOFException catch* block will never be reached because *EOFException* is a subclass of *IOException*, and therefore will match an *IOException catch* block.

Figure 11.22 shows the console output when this program is executed. Note that after we read the last object in the file and we try to read another object, the code executes the *catch* block for the *EOFException*, then the *finally* block.

Figure 11.22 Output of Example 11.18

Description

At lines 57–62, we chain several method calls to compute and output the number of flights originating from *BWI*. After creating a *Stream* from *flights* at line 59, we filter the *Stream* and keep only the objects whose origin airport is *BWI*. The call to the *filter* method at line 60 is similar to the ones in Example 11.12. Again, we can use a lambda expression because *Predicate*, the parameter of the *filter* method, is a functional interface. This time, each element of the *Stream* is a *Flight2* reference rather than a *String*. With the *flight* reference, we call the *origin* method and compare the returned value to *BWI* as our filtering criteria. At line 61, we call the *count* method to retrieve the number of elements in the *Stream* resulting from the filtering operation.

At lines 64–71, we again chain several method calls to compute and output the average number of passengers for all flights. The call to the *mapToInt* method at line 67 is similar to the call to *mapToDouble* in Example 11.12.

When we have a *Stream* of objects and we want to get an *IntStream* (or *DoubleStream*) for the *int* (or *double*) instance variable values of these objects, we specify the method of the objects’ class using the following double colon notation in place of a lambda expression:

```
ClassName::methodName
```

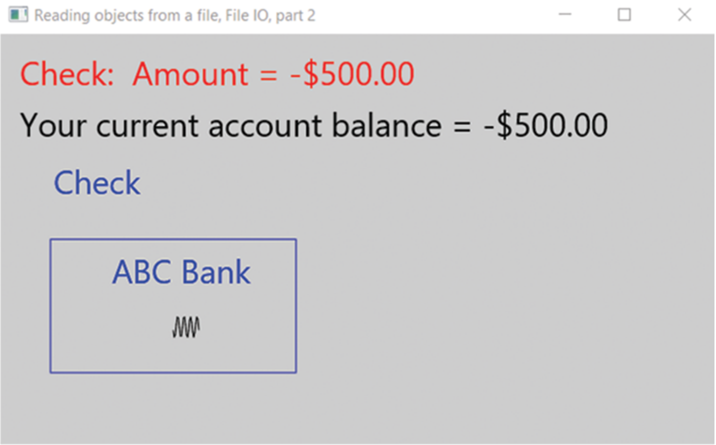
Because we want to access an *IntStream* consisting of the number of passengers for all the flights, we use the *numPassengers* method of the *Flight2* class, and we write:

```
Flight2::numPassengers
```

Thus, in order to retrieve the average number of passengers, we first retrieve the *IntStream* of number of passengers at line 67 and call the *average* method at line 68.

with these end-of-chapter questions

11.15.1	Multiple Choice Exercises Questions 10, 11
11.15.2	Reading and Understanding Code Question 16
11.15.3	Fill In the Code Question 27
11.15.6	Write a Short Program Question 49
11.15.8	Technical Writing Question 63



Check: Amount = -$500.00

Your current account balance = -$500.00

11.12 Programming Activity 2: Reading Objects from a File

In this activity, you will read objects from a file and perform this activity:

> Read an object file containing bank account transaction objects. Loop through all the objects and calculate the new balance of the bank account. Assume that we do not know the number of transaction items, that is, objects, in the file.

Notice that this activity is identical to Programming Activity 1, except that the transactions you will read are stored in the file as objects.

The framework will display the current transaction and current balance so that you can check the correctness of your code as the program executes.

For example, **Figure 11.23** demonstrates the animation: We are currently scanning a check transaction for the amount of $500.00. The original balance was $0.00 and the new balance is now –$500.00.

Figure 11.23 Animation of a $500 Check and the New Balance

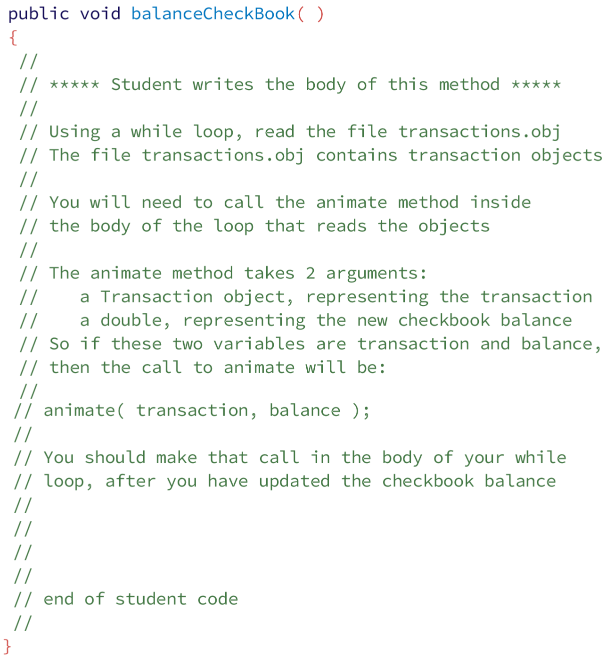
Description

Task Instructions: Reading from the transactions.obj File

In this chapter's folder in the supplied code, you will find a Programming Activity 2 folder. Copy the contents of the folder onto a folder on your computer.

- Open the *AccountingDrawing.java* file. Search for five asterisks (*****) to find the *balanceCheckBook* method where you will add your code. The method header has been coded for you. Write the code to read the transactions from the *transactions.obj* file, and calculate the balance after all the transactions in that file have been executed. This program first writes *Transaction* objects to the file *transactions.obj*; that code is provided. You need to code the body of the *balanceCheckBook* method in order to read that file. **Example 11.19** shows the student code section of the *AccountingDrawing.java* file.

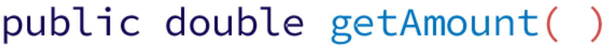
Example 11.19 The *balanceCheckBook* Method



```
public void balanceCheckBook( )
{
  //
  // ***** Student writes the body of this method *****
  //
  // Using a while loop, read the file transactions.obj
  // The file transactions.obj contains transaction objects
  //
  // You will need to call the animate method inside
  // the body of the loop that reads the objects
  //
  // The animate method takes 2 arguments:
  //    a Transaction object, representing the transaction
  //    a double, representing the new checkbook balance
  // So if these two variables are transaction and balance,
  // then the call to animate will be:
  //
 // animate( transaction, balance );
 //
 // You should make that call in the body of your while
 // loop, after you have updated the checkbook balance
 //
 //
 //
 //
 // end of student code
 //
}
```

Description

- Begin with a checkbook balance of 0.00.
- To process the transactions, you will need to read one *Transaction* object at a time from the *transactions.obj* file; you will retrieve the transaction amount using the *getAmount* method of the *Transaction* class. The API for that method is:



```
public double getAmount( )
```

Description

- Then process the transaction; you do not need to check the type of transaction. Just add the amount to the checkbook balance.
- After you have processed each transaction, call the *animate* method. This method belongs to the *Accounting* class, so you will call *animate* without using an object reference. The API of the *animate* method is the following:

```
public void animate( Transaction currentTransaction,
                     double currentBalance )
```

Description

As you can see, the *animate* method takes two arguments:

- *currentTransaction* is the current *Transaction* object
- *currentBalance* is the current balance of the checkbook

Assuming that you have a *Transaction* object reference called *transactionObject* and a *double* called *balance*, a call to *animate* will look like the following:

```
animate( transactionObject, balance );
```

Description

- When you call *animate*, the window will display the current transaction graphically. It will also display the transaction amount (red if negative, blue if positive) and the current checkbook balance (in black). By adding the previous checkbook balance to the current transaction amount, you will be able to compute the current checkbook balance and check that your program is correct.
- Stop reading from the file when you reach the end of the file. You will need to set up a *catch* block to handle the *EOFException* that occurs when the end of the file is reached.

To test your code, compile and run the *AccountingApplication.java* application file.

If you have time …

- Modify the *initialize* method of the *AccountingController* class, adding another transaction. Run the program again and verify that your code still yields the correct result. To add another transaction, you could, for instance, write this code:

```
Withdrawal w2 = new Withdrawal( -200.00 );
transactionList.add( w2 );
```

Description

You can add a transaction of type *Check, Withdrawal, Deposit,* or *Unknown-Transaction* (all of which are subclasses of the *abstract* class *Transaction*).

Troubleshooting

If your method implementation does not animate or animates incorrectly, check these items:

- Verify that you have coded the call to *animate* at the proper time.
- Verify that you have coded the condition for exiting the loop correctly.
- Verify that you have coded the body of the loop correctly.

DISCUSSION QUESTIONS

1. Explain why we cannot simply read the *transactions.obj* file as a text file.
2. Explain why we need to type cast each object that we read from the file.

```
Enter your email address > mary.jb.com
Email address does not contain @
```

```
Enter your email address > john@jb.com
Your host is jb.com
```

11.13 User-Defined Exceptions

There will be times when we want to design our own exception class because the predefined Java exception classes do not fit our needs.

Suppose we are interested in designing a class encapsulating email addresses. We will call that class *EmailAddress*. To keep things simple, we will say that a legal email address is a *String* containing the @ character. In order to prevent instantiation of objects with illegal email addresses, we will design our *EmailAddress* constructor so that it throws an exception if its argument, a *String*, does not contain the @ character.

To do that, we first design an exception class that encapsulates an illegal email exception. We call our class *IllegalEmailException* and we will *throw* an exception when the argument to the *EmailAddress* constructor does not contain the

@ character. Oracle recommends that user-defined exceptions be checked exceptions.

More generally, when a user-defined exception class is defined as a subclass of an existing Java exception class, such as *Exception*, *IOException*, or *FileNotFoundException*, our class inherits the functionality of the existing exception class, which simplifies coding the new class. We extend the *Exception* class so that our exception is checked and we can associate a specific error message with the exception. We need to code only the constructor, and the constructor's job is to pass our message to the constructor of the superclass.

Thus, the general pattern of a user-defined exception class is:

```
public class ExceptionName extends ExistingExceptionClassName
{
  public ExceptionName( String message )
  {
    super( message );
  }
}
```

Description

Example 11.20 shows our *IllegalEmailException* class.

Example 11.20 The *IllegalEmailException* Class.

```
 1 /* The IllegalEmailException class
 2    Anderson, Franceschi
 3 */
 4
 5 public class IllegalEmailException extends Exception
 6 {
 7   public IllegalEmailException( String message )
 8   {
 9     super( message );
10   }
11 }
```

Description

The constructor for the class is coded at lines 7 to 10; it takes a *String* parameter and simply passes it to the superclass constructor.

The pattern for a method that *throws* a user-defined exception is:

```
accessModifier dataType methodName( parameter list )
                        throws ExceptionName
{
  if ( parameter list is legal )
      // process the parameter list
  else
      throw new ExceptionName( "Some message here" );
}
```

Description

The message we pass to the *ExceptionName* constructor will identify the type of error we detected. When a client program catches the exception, the client can call the *getMessage* method of the exception class in order to retrieve that message.

Example 11.21 shows our *EmailAddress* class.

Example 11.21 The *EmailAddress* Class

```
 1 /* The EmailAddress class
 2    Anderson, Franceschi
 3 */
 4
 5 public class EmailAddress
 6 {
 7   public static final char AT_SIGN = '@';
 8   private String email;
 9
10   public EmailAddress( String email )
11                      throws IllegalEmailException
12   {
13     if ( email.indexOf( AT_SIGN ) != - 1 )
14          this.email = email;
15     else
16          throw new IllegalEmailException
17                ( "Email address does not contain " + AT_SIGN );
18   }
19
20   public String getHost( )
21   {
22     int index = email.indexOf( AT_SIGN );
23     return email.substring( index + 1, email.length( ) );
24   }
25 }
```

Description

We coded the constructor at lines 10–18. We test if the constructor's parameter, *email*, contains the character *AT_SIGN* (a constant equal to the @character) at line 13. If it does, we proceed normally and initialize the instance variable *email* at line 14. If it does not, we throw an *IllegalEmailException* with the appropriate message at lines 16–17. In addition to the constructor, we coded the *getHost* method at lines 20–24. The *getHost* method returns the substring comprising the characters of *email* after *AT_SIGN*. Thus, for an email address of *myEmailAddress@yahoo.com*, the *getHost* method will return *yahoo.com*.

Now that we have built our own exception class and a class including a method that *throws* that exception, we are ready to use them in a client program. This is identical to using a predefined Java exception. **Example 11.22** shows our *EmailChecker* class.

Example 11.22 The *EmailChecker* Class

```
 1 /* The EmailChecker class
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6
 7 public class EmailChecker
 8 {
 9   public static void main( String [ ] args )
10   {
11     Scanner scan = new Scanner( System.in );
12     System.out.print( "Enter your email address > " );
13     String myEmail = scan.next( );
14     try
15     {
16       EmailAddress address = new EmailAddress( myEmail );
17       System.out.println( "Your host is " + address.getHost( ) );
18     }
19     catch ( IllegalEmailException iee )
20     {
21       System.out.println( iee.getMessage( ) );
22     }
23   }
24 }
```

Description

We ask the user to input an email address, *myEmail*, at lines 12–13. We then try to instantiate the *EmailAddress* object *address* at line 16, passing *myEmail* to the constructor. If *myEmail* does not contain the @ character, our *EmailAddress* constructor *throws* an *IllegalEmailException*, which we *catch* at line 19. In this *catch* block, we print the message the *EmailAddress* constructor sent to the *IllegalEmailException* constructor. If *myEmail* contains the @ character, we continue executing inside the *try* block. **Figure 11.24** shows two runs of this example; the first generates the exception, the second completes without generating an exception.

Figure 11.24 Two Sample Runs of Example 11.22

Description

CODE IN ACTION

Within the online resources, you will find a movie with a step-by-step illustration of *try* and *catch* blocks. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

with these end-of-chapter questions

11.15.1	Multiple Choice Exercises Question 3
11.15.8	Technical Writing Question 62

© Lois GoBe/Shutterstock

CHAPTER REVIEW

11.14 Chapter Summary

- Java provides exception classes so that unexpected, illegal operations at run time can be trapped and handled. This provides the programmer with a tool to keep the program running instead of terminating.
- When calling a constructor or method that *throws* a checked exception, we must use *try* and *catch* blocks; otherwise, the code will not compile.
- For calls to a constructor or method that *throws* an unchecked exception *try* and *catch* blocks are optional. If *try* and *catch* blocks are not used, the exception will be caught at run time by the Java Virtual Machine.
- When a *try* block assigns a value to a variable and that variable is used after the *try/catch* block, the variable must be initialized before the *try* block is entered.
- A variable defined inside a *try* block is local to that block.
- The *java.io* package contains classes for input and output operations.
- In order to read from a file, that file must exist; otherwise, a *FileNotFoundException* will be thrown.
- When we open a file for writing, the file is created if it does not exist. If the file already exists, the contents of the file are deleted.

- When we open a file for appending, the file is created if it does not exist. If the file already exists, we start writing at the end of the file.
- The *Scanner* class in the *java.util* package is helpful in parsing a *String* consisting of fields separated by one or more delimiters.
- A *Record* class provides a very simple way to write a class whose only purpose is to store data.
- The *FileWriter* and *PrintWriter* classes provide functionality to write primitive data types to a text file.
- The *Stream* interface includes functionality to perform aggregate sequential and parallel operations on a collection of elements.
- We can use the *Path* and *Files* classes to convert file contents to a *Stream*. An *ArrayList* can also be converted to a *Stream*.
- Many methods of the *Stream* interface accept one parameter whose type is from a functional interface. In this case, we can use a lambda expression when calling such a method.
- We can use the *URL*, *InputStream*, and *Scanner* classes in order to read data from a remote resource.
- Data in remote sites are often formatted using XML or JSON. The *org.json* package includes classes, such as *JSONObject* and *JSONArray*, that provide the functionality to parse a JSON string.
- Objects can be written to a file; they must be instantiated from a class that *implements* the *Serializable* interface.
- The *Serializable* interface has no methods; therefore, no additional methods need to be implemented in a class that *implements* the *Serializable* interface.
- The *FileOutputStream* and *ObjectOutputStream* classes provide functionality to write objects to a file.
- To avoid writing class data to a file of objects, declare the data as *static* or *transient*, where appropriate.
- *The FileInputStream* and *ObjectInputStream* classes provide the functionality to read objects from a file.

- The *readObject* method returns the object read as an *Object* class reference. That object reference must be type cast to the appropriate class.
- To define our own exception, we create a class that extends an existing exception class. This class will consist of a constructor that accepts a message and passes the message to the superclass constructor.
- The method that will generate the exception includes the *throws* clause in the method header. If the invalid condition is detected, the method *throws* a new object of the user-defined exception.

11.15 Exercises, Problems, and Projects

11.15.1 Multiple Choice Exercises

1. Why are *try/catch* blocks useful?

    - They can replace selection statements, thus saving CPU time.
    - *try/catch* blocks enable programmers to attempt to recover from illegal situations and continue running the program.

4. Some methods that *throw* an exception require *try* and *catch* blocks, while some do not.

    - true
    - false

7. What keyword is found in the header of a method that could detect an error and generate an appropriate exception?

    - *throw*
    - *throws*
    - *exception*
    - *exceptions*

12. When coding a *try* and *catch* block, it is mandatory to code a *finally* block.

    - true
    - false

15. Most input- and output-related classes can be found in the package

- *java.file*
- *java.inputoutput*
- *java.io*
- *java.readwrite*

20. If we open a file for reading and the file does not exist,

    - there is a compiler error.
    - an exception is thrown.
    - the file will be created automatically.

24. When we open a file for writing,

    - we will be adding data at the end of the file.
    - the contents of the file, if any, will be deleted.
    - there is a run-time error if the file does not exist.

28. When we open a file for appending,

    - we will be adding data at the end of the file.
    - the contents of the file, if any, will be deleted.
    - there is a run-time error if the file does not exist.

32. What is the keyword for a class whose purpose is only to store data?

    - Object
    - Record
    - Data

36. What method can we use to convert an *ArrayList* to a *Stream*?

    - *convert*
    - *stream*
    - *toStream*

40. Which interface must be implemented by a class whose objects will be written to a file directly?

   - none
   - *Serializable*
   - *IO*
   - *Object*

11.15.2 Reading and Understanding Code

1. Assuming the file *words.txt* holds the following data:
   CS1 Java Illuminated
   what is the output of this code sequence:
   a. if the file is found?
   b. if the file is not found?

```
 try
 {
   Scanner file = new Scanner( new File( "words.txt" ) );

   String result = "";

   while ( file.hasNext( ) )
   {
     String s = file.next( );

     result += s;
     result += " AND ";
   }
   System.out.println( "result is " + result );
   file.close( );

 }
catch ( FileNotFoundException fnfe )
{
   System.out.println( "Unable to find words.txt" );
}
catch ( IOException ioe )
{
   ioe.printStackTrace( );
}
```

   Description

2. What is the output of this code sequence?

```
Scanner parse = new Scanner( "A B C D" );
while ( parse.hasNext( ) )
   System.out.print( parse.next( ) );
```

Description

3. What is the output of this code sequence?

```
Scanner parse = new Scanner( "AA:BB:CC" );
parse.useDelimiter( ":" );
while ( parse.hasNext( ) )
    System.out.println( parse.next( ) );
```

Description

4. What is the output of this code sequence?

```
Scanner parse = new Scanner( "oneANDtwoANDthreeANDfour" );
parse.useDelimiter( "AND" );
while ( parse.hasNext( ) )
   System.out.println( parse.next( ) );
```

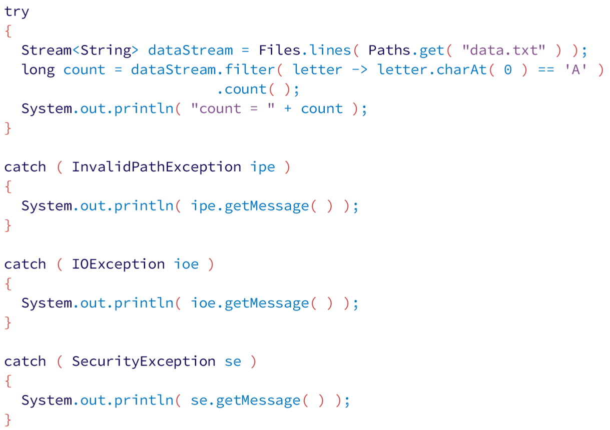
Description

For Questions 16, 17, 18, and 19, you should assume that the file *data.txt* contains the following text:

```
A
B
C
A
B
A
```

Description

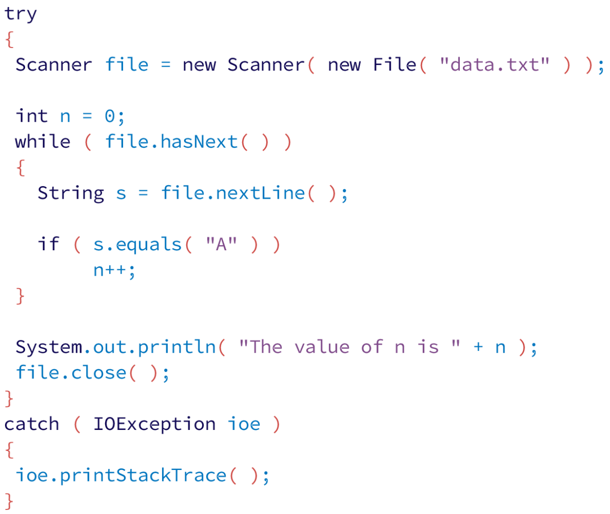
1. What is the output of this code sequence?



```
try
{
  Stream<String> dataStream = Files.lines( Paths.get( "data.txt" ) );
  long count = dataStream.filter( letter -> letter.charAt( 0 ) == 'A' )
                         .count( );
  System.out.println( "count = " + count );
}

catch ( InvalidPathException ipe )
{
  System.out.println( ipe.getMessage( ) );
}

catch ( IOException ioe )
{
  System.out.println( ioe.getMessage( ) );
}

catch ( SecurityException se )
{
  System.out.println( se.getMessage( ) );
}
```

Description

2. What is the output of this code sequence?



```
try
{
 Scanner file = new Scanner( new File( "data.txt" ) );

 int n = 0;
 while ( file.hasNext( ) )
 {
   String s = file.nextLine( );

   if ( s.equals( "A" ) )
        n++;
 }

 System.out.println( "The value of n is " + n );
 file.close( );
}
catch ( IOException ioe )
{
 ioe.printStackTrace( );
}
```

Description

3. What is the output of this code sequence?

```
try
{
  Scanner file = new Scanner( new File( "data.txt" ) );

  while ( file.hasNext( ) )
  {
    String s = file.nextLine( );
    if ( s.equals( "A" ) )
         System.out.println( "Excellent" );
    else if ( s.equals( "B" ) )
         System.out.println( "Good" );
    else
         System.out.println( "Try to do better" );
  }
  file.close( );
}
catch ( IOException ioe )
{
  ioe.printStackTrace( );
}
```

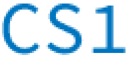
Description

4. What is the output of this code sequence?

```
try
{
 Scanner file = new Scanner( new File( "data.txt" ) );

 String s = "";
 while ( file.hasNext( ) )
 {
   s = file.nextLine( );
 }

 if (  s.equals( "A" ) )
   System.out.println( "Nice finish" );
  file.close( );
 }
 catch ( IOException ioe )
 {
   ioe.printStackTrace( );
 }
```

Description

5. The file *data.txt* contains the following text:



```
CS1
```

What does the file *data.txt* contain after this code sequence is executed?

```
try
{
 FileWriter fw = new FileWriter( "data.txt", true );
 PrintWriter pw = new PrintWriter( fw );

 pw.println( "Java Illuminated" );
 pw.close( );
}
catch ( IOException ioe )
{
 ioe.printStackTrace( );
}
```

Description

6. The file *data.txt* contains the following text:

```
CS1
```

What does the file *data.txt* contain after this code sequenced is executed?

```
try
{
 FileWriter fw = new FileWriter( "data.txt", false );
 PrintWriter pw = new PrintWriter( fw );

 String s = "ABCDEFGH";

 for ( int i = 0; i < s.length( ); i++ )
 {
   if ( i % 2 == 0 )
     pw.print( s.charAt( i ) );
 }
 pw.println( );
 pw.close( );
}
catch ( IOException ioe )
{
 ioe.printStackTrace( );
}
```

Description

7. The file *data.txt* contains the following text:

```
CS1
```

What does the file contain after the following code sequence is executed?

```
try
{
 FileWriter fw = new FileWriter( "data.txt", true );
 PrintWriter pw = new PrintWriter( fw );

 for ( int i = 0; i < 5; i++ )

     pw.println( i );
  pw.close( );
 }
 catch ( IOException ioe )
 {
   ioe.printStackTrace( );
 }
```

Description

8. What does the file *data.txt* contain after the following code sequence is executed?

```
try
{
 FileWriter fw  = new FileWriter( "data.txt", false );
 PrintWriter pw = new PrintWriter( fw );

 int s = 0;
 for ( int i = 0; i < 5; i++ )
 {
   s += i;
 }
 pw.print( "The result is " );
 pw.print( s );
 pw.close( );
}
catch ( IOException ioe )
{
  ioe.printStackTrace( );
}
```

Description

11.15.3 Fill In the Code

1. This code segment reads a file named *data.txt* that contains one data item per line, and outputs only the data items that are integers. Hint: You may need to use nested *try* and *catch* blocks.

```
String s = "";
int n = 0;
try
{
   Scanner file = new Scanner( new File( "data.txt" ) );
   while ( file.hasNext( ) )
   {
      s = file.nextLine( );
      // your code goes here
   }
}
catch ( IOException ioe )
{
 ioe.printStackTrace( );
}
```

Description

2. This code retrieves the "*C* " in the string "*A-B-C-D*" using *Scanner* and outputs it:

```
Scanner parse = new Scanner( "A-B-C-D" );
parse.useDelimiter( "-" );
String s = "";
// your code goes here

System.out.println( s );
```

Description

For Questions 26–29, you should assume that the file *data.txt* contains the following:

Java Illuminated: Programming Is Not A Spectator Sport

Description

3. This code sequence reads the first two lines of the file *data.txt* and outputs them to the console.

```
try
{
  Scanner file = new Scanner( new File( "data.txt" ) );
  // your code goes here
```

Description

4. This code sequence reads the file *data.txt* and outputs the lowest *String* in lexicographic order to the console. Assume that *data.txt* contains one word per line.

```
try
{
  Stream<String> dataStream = Files.lines( Paths.get( "data.txt" ) );
  // your code goes here

}
catch ( InvalidPathException ipe )
{
  System.out.println( ipe.getMessage( ) );
}
catch ( IOException ioe )
{
  System.out.println( ioe.getMessage( ) );
}
catch ( SecurityException se )
{
  System.out.println( se.getMessage( ) );
}
```

Description

5. This code sequence reads the file *data.txt*, concatenates all the lines with a space between them, and outputs them as:

```
Java Illuminated: Programming Is Not A Spectator Sport
```

```
try
{
  Scanner file = new Scanner( new File( "data.txt" ) );

  String result = "";
  // your code goes here

}
catch ( IOException ioe )
{
  ioe.printStackTrace( );
}
```

Description

6. This code sequence reads the file *data.txt* and outputs only the lines that start with the *String* “*Sp*”. Assume that we do not know the contents of the file before reading it.
For the current example, the output will be:

```
Spectator
Sport

try
{
  Scanner file = new Scanner( new File( "data.txt" ) );

  String result = "";
  // your code goes here

}
catch ( IOException ioe )
{
  ioe.printStackTrace( );
}
```

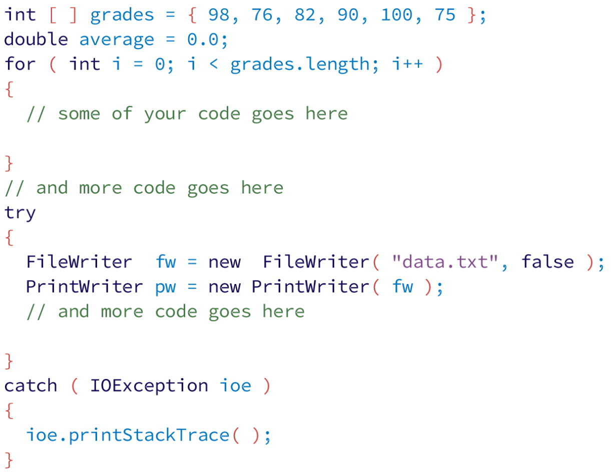
Description

7. This code sequence loops through the array *grades* and writes all its elements to the file *data.txt*, one per line:

```
int [ ] grades = { 98, 76, 82, 90, 100, 75 };
try
{
  FileWriter fw = new FileWriter( "data.txt", false );
  // your code goes here

}
// and your code continues here
```

Description

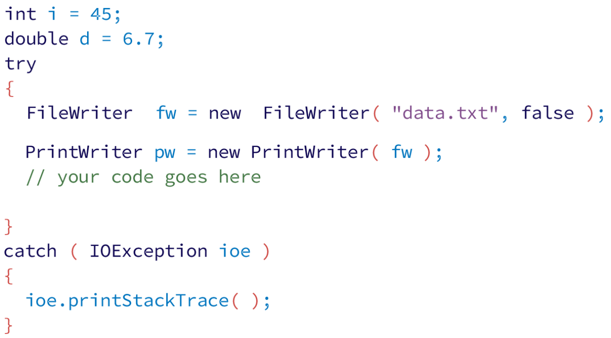
8. This code sequence loops through the array *grades*, calculates the average, and writes the average to the file *data.txt*:



```
int [ ] grades = { 98, 76, 82, 90, 100, 75 };
double average = 0.0;
for ( int i = 0; i < grades.length; i++ )
{
  // some of your code goes here

}
// and more code goes here
try
{
  FileWriter  fw = new  FileWriter( "data.txt", false );
  PrintWriter pw = new PrintWriter( fw );
  // and more code goes here

}
catch ( IOException ioe )
{
  ioe.printStackTrace( );
}
```

Description

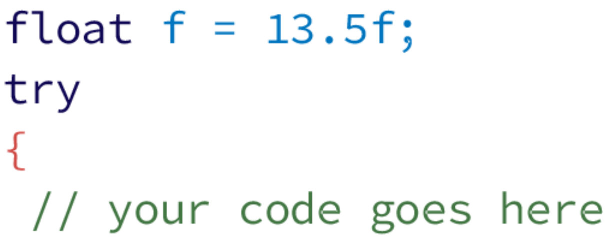
9. This code sequence writes the values of the variables *i* and *d* to the file *data.txt*, one line at a time.



```
int i = 45;
double d = 6.7;
try
{
  FileWriter  fw = new  FileWriter( "data.txt", false );

  PrintWriter pw = new PrintWriter( fw );
  // your code goes here

}
catch ( IOException ioe )
{
  ioe.printStackTrace( );
}
```

Description

10. This code sequence appends the value of the variable *f* to the file *data.txt*:



```
float f = 13.5f;
try
{
  // your code goes here
```

Description

11.15.4 Identifying Errors in Code

1. Where is the error in this code sequence?

   ```
   Scanner parse = new Scanner( "1 2 3" );
   int i = parse.next( );
   ```

   Description

2. Where is the error in this code sequence?

   ```
   try
   {
     Scanner file = new Scanner( new File( "data.txt" ) );
     String s = file.nextLine( );
   }
   catch ( ArithmeticException ae )
   {
     System.out.println( ae.getMessage( ) );
   }
   ```

   Description

3. Where is the error in this code sequence?

   ```
   try
   {
     Scanner file = new Scanner( new File( "data.txt" ) );
     file.write( "Hello" );
   }
   catch ( IOException ioe )
   {
     ioe.printStackTrace( );
   }
   ```

   Description

11.15.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. You coded the following in the class *Test.java*:

```
import java.io.IOException;
import java.io.File;
import java.util.Scanner;
public class Test
{
  public static void main( String [ ] args )
  {
      try // line 9
      {

            Scanner file = new Scanner( new File( "data.txt" ) );

            String stringRead = file.nextLine( );
            System.out.println( stringRead );
      }
  }
}
```

Description

At compile time, you get the following error:

```
Test.java:9: error: 'try' without 'catch', 'finally' or resource
declarations
   try    // line 9
   ^
1 error
```

Description

Explain what the problem is and how to fix it.

2. You coded the following in the class *Test.java*:

```
import java.io.IOException;
import java.io.File;
import java.util.Scanner;

public class Test
{
  public static void main( String [ ] args )
  {
      try
      {
       Scanner file = new Scanner( new File( "data.txt" ) );

       String stringRead = file.nextLine( );
      }
      catch ( IOException ioe )
      {
       ioe.printStackTrace( );
      }
      System.out.println( "string read: " + stringRead );
      // line above is 19
   }
}
```

Description

At compile time, you get the following error:

```
Test.java:19: error: cannot find symbol
   System.out.println( "string read: " + stringRead );
                                         ^
symbol:   variable stringRead
location: class Test
1 error
```

Explain what the problem is and how to fix it.

3. You coded the following in the class *Test.java*:

```
import java.io.IOException;
import java.io.File;
import java.util.Scanner;
public class Test
{
  public static void main( String [ ] args )
  {
    String stringRead;
    try
    {
     Scanner file = new Scanner( new File( "data.txt" ) );

     stringRead = file.nextLine( );
    }
    catch ( IOException ioe )
    {
     ioe.printStackTrace( );
    }
    System.out.println( "string read: " + stringRead );
    // line above is line 21
 }
}
```

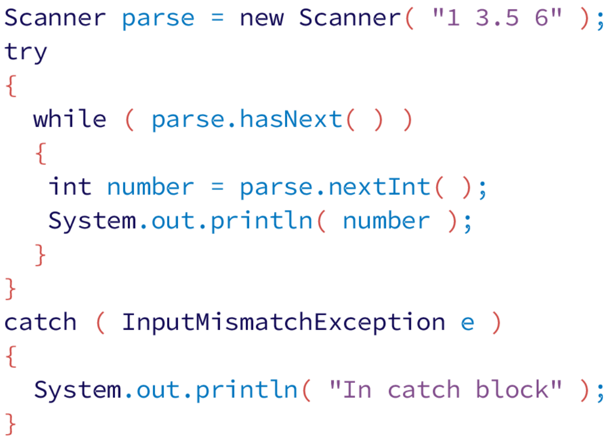
At compile time, you get the following error:

```
Test.java:21: error: variable stringRead might not have been
initialized
System.out.println( "string read: " + stringRead );
                                      ^
1 error
```

Description

Explain what the problem is and how to fix it.

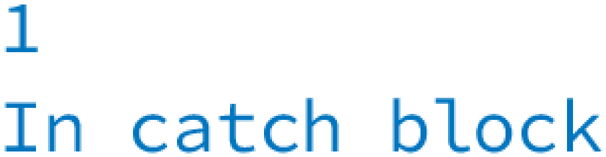
4. You coded the following in the class *Test.java*:



```
Scanner parse = new Scanner( "1 3.5 6" );
try
{
  while ( parse.hasNext( ) )
  {
   int number = parse.nextInt( );
   System.out.println( number );
  }
}
catch ( InputMismatchException e )
{
  System.out.println( "In catch block" );
}
```

Description

The code compiles and runs, but the result is:

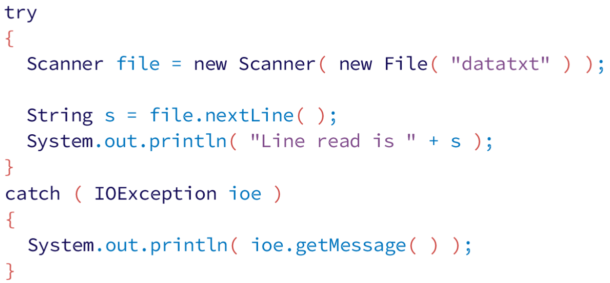


```
1
In catch block
```

Description

Explain why the *catch* block executes and change the data so that all the numbers are processed.

5. In order to read from the file *data.txt*, you coded the following in the *Test.java* class:



```
try
{
  Scanner file = new Scanner( new File( "datatxt" ) );

  String s = file.nextLine( );
  System.out.println( "Line read is " + s );
}
catch ( IOException ioe )
{
  System.out.println( ioe.getMessage( ) );
}
```

Description

The code compiles and runs, but here is the output:

```
datatxt (The system cannot find the file specified)
```

Description

Explain what the problem is and how to fix it.

6. You coded the following in the class *Test.java*:

```
import java.io.*;

public class Test
{
 public static void main( String [ ] args )
 {
   FileWriter fw = new FileWriter( "data.txt", false );
   PrintWriter pw = new PrintWriter( fw );

   pw.println( "hi" );
   pw.close( );
 }
}
```

Description

At compile time, you get the following error:

```
Test.java:7: error: unreported exception IOException; must be
caught or declared to be thrown
   FileWriter  fw = new  FileWriter ( "data.txt", false);
                    ^
1 error
```

Description

Explain what the problem is and how to fix it.

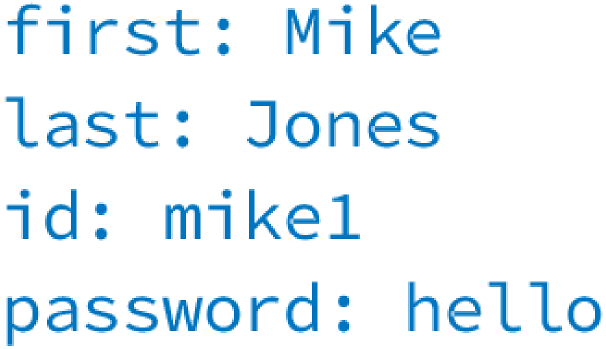
11.15.6 Write a Short Program

1. Write a program that reads a file and writes a copy of the file to another file with line numbers inserted.
2. In Internet programming, programmers receive parameters via a query string, which looks like a *String* with fields separated by the “&” character. Each field typically has a metadata part that identifies the data followed by an equals sign and then the data. An example of a query string is:

```
first=Mike&last=Jones&id=mike1&password=hello
```

Description

Using *Scanner* at least once, parse a query string and output each field on a different line after replacing the equal sign with a colon followed by a space. For example, for the preceding sample query string, the output should be:



```
first: Mike
last: Jones
id: mike1
password: hello
```

Description

3. Write a program that reads a file that contains only one line; output all the characters, one character per line.
4. Write a program that reads a file and counts how many lines it contains.
5. Write a program that reads a text file that contains a grade (for instance, 87) on each line. Calculate and print the average of the grades.
6. Write a program that reads a text file and writes to a file every line of the file separated by a blank line.
7. Modify Example 11.15 and output to the console the number of movies that start with a given character. Allow the user to input a character and output the number of movies that start with that character.
8. Often websites display the visitor count ("You are visitor number 5246"). Write a program that reads a file that holds the visitor count, outputs it, and updates the file, incrementing the visitor count by 1.

9. Often on websites, the beginning of an article is displayed followed by the word more and several dots (as in *more*…). Write a graphical application that reads the first two lines of a file, and displays them inside a window, adding the word *more* in blue followed by three dots.

11.15.7 Programming Projects

1. Design a class that checks if a *String* is made of tokens of the same data type (for this, you may only consider four data types: *boolean, int, double*, or *char*). This class has two instance variables: the *String* of data and its delimiter. Other than the constructor, you should include a method, *checkTokens,* that takes one parameter, representing the data type expected (for example, 0 could represent *boolean*, 1 could represent *int*, 2 could represent *double*, and 3 could represent *char*). The method returns *true* if all the tokens in the *String* are of the specified data type; otherwise, it returns *false*.
2. We are interested in checking the number of times a given word (for example, the word *secret*) appears in a file. You should assume that lines do not wrap, that is, a line does not continue on the next line. Warning: You could have letters arranged like *secsecret*. Design a class that encapsulates that idea. Test it with a client program.
3. Design a class that checks if the contents of two text files are identical and, if not, determines how many lines are different. Lines are different if they differ in one or more characters. Test your class with a client program.

4. Design a class that encapsulates the contents of a text file. Include the following methods in your class: *numberOfLinesInFile*, *longestLineInFile* (the line number of the line containing the maximum number of characters), *shortestLineInFile*, and *averageNumberOfCharactersPerLine*. Test your class with a client program.
5. Alter Example 11.7 so that *Scanner* will use a comma and semicolon as delimiters. Hint: you will need to use a *Pattern*.
6. A file contains web addresses, one on each line. Design a class that encapsulates the concept of counting the number of college addresses (contains *.edu*), government addresses (contains *.gov*), business addresses (contains *.com*), organization addresses (contains *.org*), or other addresses. Test your class with a client program.
7. In cryptograms, each character is encoded into another. If

the text is long enough, one can, as a strategy, use the frequency of occurrence of each character. The most frequently occurring character will likely be the code for an *e*, because *e* is the most frequently used letter of the English alphabet. Design a class that attempts to determine the relative frequency of each letter by reading a file and keeping track of the number of times each of the 26 English alphabet characters appears. Also provide methods, such as *highestFrequencyCharacter* and *lowestFrequencyCharacter*. Test your class with a client program.

8. Design a class that calculates statistics on data in a file. We expect the file to contain grades represented by integer values, one per line. If you encounter a value that is not an integer, you should throw an exception, print a message to the console, skip that value, and continue processing. Store the grades that you read in an *ArrayList* so that all the grades are available for retrieval. You should also have, as a minimum, methods that return the grade average, the highest grade, the lowest grade, and ones that return all the grades as an array of letter grades. Test your class with a client class.
9. Write a *Record* class encapsulating the concept of a home, assuming that it has the following attributes: the number of rooms, the square footage, and whether it has a basement. Write a client program that creates five *Home* objects, writes them to a file as objects, then reads them from the file as objects, outputs a description of each object using the *toString* method, and outputs the number of *Home* objects. When reading the objects, you should assume that you do not know the number of objects in the file.

10. Using *Streams*, output the number of the following in a Java file: constructors, *public* variables or methods, *private* variables or methods, *static* variables or methods. Assume that the words *public*, *private*, and *static* can appear at most once per line. Also output if the number of opening curly braces is equal to the number of closing curly braces. Also assume that there can be only one opening or closing curly brace per line at the most.

11.15.8 Technical Writing

1. Are exceptions a good thing or a bad thing? Argue both sides.
2. With respect to writing objects to and reading objects from a file, discuss the importance of documenting your code well.

11.15.9 Group Project (for groups of 2, 3, or more students)

1. A friend of yours owns two houses at Football City, the site of the next Super Bowl. Your friend wants to rent those two houses for the Friday, Saturday, and Sunday of the Super Bowl weekend. House #1 has 3BR (3 bedrooms), 3BA (3 baths), and house #2 has 1BR, 1BA.

   For this project, concurrency is not an issue; you should assume that two customers will never access your system at exactly the same time. You should also assume that the management-sid014"TX">This friend has asked you to build a file-based reservation system enabling the following:

   A. Management-side software:

   Your friend controls the rental price and may change it every day. He/she sends you a change file every day; this file may be empty, in which case there are no pricing changes. If the file is not empty, pricing has changed (for one or more houses, or for one or more days). You are in charge of this project, and therefore, you are in charge of specifying the file format; however, this must be a simple text file because your friend is not a computer person.

You do not have to simulate the act of sending the file by your friend; you should assume that the file is a text file in your directory and that you only need to read the data.

Your management-side software needs to read this file and update a different file, with which you control the reservation system. You can create your own design for the structure of that file. Of course, prices for existing reservations cannot be changed.

Finally, your management-side software should write to a file the status of the reservations; that is, which house is rented to whom, when, and for what price.

B. Customer-side software:

The customer-side software allows a customer to make a reservation. You should prompt the customer for a possible reservation, offering whatever house is available, when, and at what price. Do not offer a customer a house that is already rented.

In this simple version, a customer makes and pays for the reservation at the same time. Also, a reservation cannot be cancelled. When a reservation is made, the customer-side software automatically updates the file controlling the reservations.

© Lois GoBe/Shutterstock.

CHAPTER 12

Recursion

CHAPTER CONTENTS

Introduction
12.1 Simple Recursion: Identifying the General and Base Cases
12.2 Recursion with a Return Value
    12.2.1 Computing the Factorial of a Number
    12.2.2 Computing the Greatest Common Divisor
12.3 Recursion with Two Base Cases
12.4 **Programming Activity 1: Checking for a Palindrome**
12.5 Simple Recursion on an Array
12.6 Binary Search: A Recursive Solution
12.7 **Programming Activity 2: The Towers of Hanoi**
12.8 Merge Sort
12.9 Quick Sort
12.10 Recursion Versus Iteration
12.11 Chapter Summary
12.12 Exercises, Problems, and Projects
    12.12.1 Multiple Choice Exercises
    12.12.2 Reading and Understanding Code
    12.12.3 Fill In the Code

12.12.4 Identifying Errors in Code

12.12.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

12.12.6 Write a Short Program

12.12.7 Programming Projects

12.12.8 Technical Writing

12.12.9 Group Projects

Introduction

Small problems are easier to solve than big ones, with or without the help of a computer. For example, it is easy to see that 14 is a multiple of 7, but determining if 12,348 is a multiple of 7 requires some thinking . . . or a well-programmed computer.

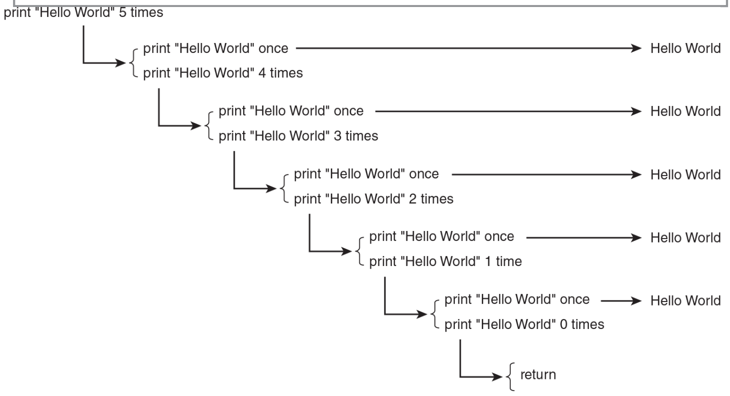
If we knew that 12,341 is a multiple of 7, then it would be easy to determine that 12,348 is also a multiple of 7, because 12,348 is simply 12,341 + 7. But then, it is not that easy to determine that 12,341 is a multiple of 7. But again, if we knew that 12,334 is a multiple of 7, then it would be easy to determine that 12,341 is also a multiple of 7, because 12,341 is simply 12,334 + 7. Well, if we keep subtracting 7 from the current number, eventually, either we will arrive at 0, which means that 12,348 is a multiple of 7, or we will arrive at a number less than 7 but not 0, which means that 12,348 is not a multiple of 7. Thus, we have reduced a large problem to a small problem that is easy to solve.

The idea of **recursion** is to reduce the size of a problem at each step so that we eventually arrive at a very small, easy-to-solve problem. That easy-to-solve problem is called the **base case**. The formula that reduces the size of the problem is called the **general case**. The general case takes us from solving a bigger problem to solving a smaller problem.

A method that uses recursion calls itself. In other words, in the body of a **recursive method**, there is a call to the method itself. The arguments passed are smaller in value (that is, they get us closer to the base case) than the original arguments. The recursive method will keep calling itself with arguments that are smaller and smaller in value, until eventually we reach the base case.

Any problem that can be solved recursively can also be solved using a loop, or iteration. Often, however, a recursive solution to a problem provides simpler, more elegant, and more compact code than its iterative counterpart.

```
Hello World
Hello World
Hello World
Hello World
Hello World
```


print "Hello World" 5 times
print "Hello World" once
print "Hello World" 4 times
Hello World
print "Hello World" once
print "Hello World" 3 times
Hello World
print "Hello World" once
print "Hello World" 2 times
Hello World
print "Hello World" once
print "Hello World" 1 time
Hello World
print "Hello World" once
print "Hello World" 0 times
Hello World
return

12.1 Simple Recursion: Identifying the General and Base Cases

When designing a recursive solution for a problem, we need to do two things:

- define the base case
- define the rule for the general case

For example, if we want to print “Hello World” 100 times, we can do the following:

- print “Hello World” once
- print “Hello World” 99 times

Note that we do two things: First, we print “Hello World” once, which is easy to do. Then we reduce the size of the remaining problem to printing “Hello World” 99 times. In order to print “Hello World” 99 times, we print “Hello World” once, then we print “Hello World” 98 times. Continuing the same approach, to print “Hello World” 98 times, we print “Hello World” once, then we print “Hello World” 97 times, and so on. Eventually, we will reach a point where we print “Hello World” once, then print “Hello World” 0 times. Printing “Hello World” 0 times is an easy-to-solve problem; we simply do nothing. That is our base case for this problem.

Thus, our general approach to printing “Hello World” *n* times (where *n* is greater than 0) is to print “Hello World” once, and then print “Hello World” *n* 2 1 times. As we reduce the number of times we print “Hello World,” we will eventually reach 0, the base case. This condition is easy to detect. Thus, we can solve the large problem by reducing the problem to smaller and smaller problems until we find a problem that we know how to solve.

The following pseudocode illustrates the approach for our recursive method.

```
void printHelloWorldNTimes( int n )
{
   if ( n is greater than 0 )
   {
       print "Hello World"
       printHelloWorldNTimes( n - 1 )
   }
   // else do nothing
}
```

Description

When *n* is greater than 0, we will execute the body of the *if* statement, printing “Hello World” once, then printing it *n* 2 1 times. This is the general case for this problem. We can see that we are going from a problem of size *n* (print “Hello World” *n* times) to a problem of size (*n* 2 1) (print “Hello World” *n* 2 1 times).

When *n* is 0 (or less), we do nothing; that is, the call to *printHelloWorldNTimes* with an argument of 0 does not generate any action. This is the base case, and this is when the recursive calls will end.

Example 12.1 shows this method.

Example 12.1 Recursively Printing “Hello World” *n* Times

```
 1  /* Printing Hello World n times using recursion
 2     Anderson, Franceschi
 3  */
 4
 5  public class RecursiveHelloWorld
 6  {
 7    public static void main( String [ ] args )
 8    {
 9       // print "Hello World" 5 times using our recursive method
10       printHelloWorldNTimes( 5 );
11    }
12
13    // the recursive method
14    public static void printHelloWorldNTimes( int n )
15    {
16     if ( n > 0 )
17     {
18      // print "Hello World" once
19      System.out.println( "Hello World" );
20
21      // now print "Hello World" ( n - 1 ) times
22      printHelloWorldNTimes( n - 1 );
23     }
24     // if n is 0 or less, do nothing
25    }
26  }
```

Description

We coded the *printHelloWorldNTimes* method from line 13 to line 25. That method prints “Hello World” *n* times, where *n* is an *int*, the only parameter of the method. We test at line 16 for the general case: *n* is greater than 0. There is no *else* clause: if *n* is 0 or less, we have reached the base case and the method does nothing.

The code for the general case is executed at lines 18–22. At line 19, we print “Hello World” once. At line 22, we make a recursive call to the *printHelloWorldNTimes* method in order to print “Hello World” (*n* 2 1) times. The method calls itself, but with an argument that is 1 less than its argument *n*.

SOFTWARE ENGINEERING TIP

If the method does nothing in the base case, it is important to document that fact to show when the recursive calls will end.

On line 10, we call the *printHelloWorldNTimes* method, passing the argument 5. Because *main* is *static*, it can call only *static* methods; therefore, we need to define our *printHelloWorldNTimes* method as *static*. In general, recursive methods can be defined as *static* or *nonstatic*.

Figure 12.1 shows the output of Example 12.1. As we can see, “Hello World” is indeed printed five times. **Figure 12.2** illustrates how the recursive calls are executed and the output resulting from the calls.

Figure 12.1 Output of Example 12.1

Description

Figure 12.2 Recursive Method Calls

Description

with these end-of-chapter questions

12.12.1	Multiple Choice Exercises Questions 1, 3, 4, 5
12.12.2	Reading and Understanding Code Questions 16, 17, 18, 19
12.12.6	Write a Short Program Questions 39, 40, 41, 44

```
Factorial( 5 ) is 120
```

```
factorial( 5 ) called
   Current Stack
      14 main
factorial( 4 ) called
   Current Stack
      46 factorial
      14 main
factorial( 3 ) called
   Current Stack
      46 factorial
      46 factorial
      14 main
factorial( 2 ) called
   Current Stack
      46 factorial
      46 factorial
      46 factorial
      14 main
factorial( 1 ) called
   Current Stack
      46 factorial
      46 factorial
      46 factorial
      46 factorial
      14 main
factorial( 0 ) called
   Current Stack
      46 factorial
      46 factorial
      46 factorial
      46 factorial
      46 factorial
      14 main

Base case detected

factorial( 0 ) returning 1
   Current Stack
      46 factorial
      46 factorial
      46 factorial
      46 factorial
      46 factorial
      14 main
factorial( 1 ) returning 1 * 1
   Current Stack
      46 factorial
      46 factorial
      46 factorial
      46 factorial
      14 main
factorial( 2 ) returning 2 * 1
   Current Stack
      46 factorial
      46 factorial
      46 factorial
      14 main
factorial( 3 ) returning 3 * 2
   Current Stack
      46 factorial
      46 factorial
      14 main
factorial( 4 ) returning 4 * 6
   Current Stack
      46 factorial
      14 main
factorial( 5 ) returning 5 * 24
   Current Stack
      14 main

Factorial( 5 ) is 120
```

```
Exception in thread "main" java.lang.StackOverflowError
  at RecursiveFactorial.factorial(RecursiveFactorial.java:23)
  at RecursiveFactorial.factorial(RecursiveFactorial.java:23)
  at RecursiveFactorial.factorial(RecursiveFactorial.java:23)
  at RecursiveFactorial.factorial(RecursiveFactorial.java:23)
```

```
The GCD of 474 and 162 is 6
```

```
gcd( 474, 162 )   474 % 162 = 150
   Current stack
      13 main
gcd( 162, 150 )   162 % 150 = 12
   Current stack
      48 gcd
      13 main
gcd( 150, 12 )   150 % 12 = 6
   Current stack
      48 gcd
      48 gcd
      13 main
gcd( 12, 6 )   12 % 6 = 0
   Current stack
      48 gcd
      48 gcd
      48 gcd
      13 main

base case reached, returning 6
   Current stack
      48 gcd
      48 gcd
      48 gcd
      13 main
gcd( 12,6 } returning 6
   Current stack
      48 gcd
      48 gcd
      13 main
gcd( 150,12 } returning 6
   Current stack
      48 gcd
      13 main
gcd( 162,150 } returning 6
   Current stack
      13 main

The GCD of 474 and 162 is 6
```

12.2 Recursion with a Return Value

In the preceding example, we coded a very simple method. Now let’s look at some examples that are a little more complex, with recursive methods that return a value.

In a value-returning method, the *return* statement can include a call to another value-returning method, as in:

```
public static int multiplyAbsoluteValueBy3( int n )
{
   return ( 3 * Math.abs( n ) );
}
```

Description

In this case, the *multiplyAbsoluteValueBy3* method cannot return its value until the *abs* method returns a value, allowing the expression in the *return* statement to be fully evaluated.

The same principle applies to a value-returning method that is recursive. The return value of the recursive method often consists of an expression that includes a call to the method itself.

Thus, in the general case of the method, we could see code like:

```
  return ( expression including a recursive call to the method );
```

Description

Each execution of the recursive method must wait to return its value until its recursive call to the method returns a value. When the base case is reached, the method simply returns a value without making another recursive call. At that point, the method that invoked the method with the base case argument receives its return value, which allows that method to return a value to its caller, and so on, until the method is able to return a value to the initial caller. In this way, the return values unravel up to the initial caller.

To see how this works, let's look at an example of a recursive method that returns a value.

12.2.1 Computing the Factorial of a Number

We will define a recursive method to compute and return the factorial of a positive integer.

The factorial of a positive number is defined as follows:

$$\text{factorial}(n) = n! = n * (n-1) * (n-2) * (n-3) * \ldots * 2 * 1$$

Description

The factorial of a negative number is not defined. The factorial of 0, by convention is 1.

$$\text{factorial}(0) = 0! = 1$$

Description

Let's define the base case and the general case for computing the factorial of a number.

In order to define the rule for the general case, we need to find a relationship between the problem at hand (computing the factorial of a number *n*), and a smaller, similar problem, involving, for example, (*n* 2 1), (*n* 2 2), or other smaller values of *n*. So, here we will try to establish a relationship between factorial(*n*) and factorial(*n* 2 1), factorial(*n* 2 2), and so on.

Let's first examine what the value of factorial(*n* 2 1) is. Applying the preceding formula, we get:

$$\text{factorial}(n-1)=(n-1)!=(n-1)*(n-2)*(n-3)*\ldots*2*1$$

Description

As we can see from the preceding formulas, there is a very simple relationship between factorial(*n*) and factorial(*n* 2 1):

$$\text{factorial}(n)=n*\text{factorial}(n-1)$$

Description

This is the relationship we will use for the formulation of the general case.

Using this formula, at each step we reduce the size of the problem (measured by the value of the input *n*) from *n* to (*n* 2 1). In order to compute factorial(*n*), we will call the factorial method with the argument (*n* 2 1) and multiply the returned value by *n*. The call to the factorial method with the argument (*n* 2 1) will generate a recursive call to the factorial method with argument (*n* 2 2), until eventually we generate a recursive call to the factorial method with the argument 0. We know how to compute factorial (0): by convention, it is 1. That is our base case and we have reached it. We will return 1, which will allow the unraveling of the recursive method calls until we solve the original problem, factorial(*n*).

Example 12.2 shows the code for calculating a factorial recursively. In order to keep things simple, we will also return 1 if the argument sent to the method is negative. However, we are careful in documenting our method to emphasize that the argument should be greater than or equal to 0. If we do not want to return anything when a negative argument is passed to the method, we would need to throw an exception, because the method is a value-returning method.

Example 12.2 Computing a Factorial Using Recursion

```
 1 /* Computing the factorial of a number using recursion
 2    Anderson, Franceschi
 3 */
 4
 5 public class RecursiveFactorial
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     // compute factorial of 5 and output it
10     System.out.println( "Factorial( 5 ) is "
11                          + factorial( 5 ) );
12   }
13
14   /** recursive factorial method
15   *   @param    n  a positive integer
16   *   @return      the factorial of n
17   */
18   public static int factorial( int n )
19   {
20     if ( n <= 0 ) // base case
21       return 1;
22     else                // general case
23       return ( n * factorial( n - 1 ) );
24   }
25 }
```

Description

Here is how our *factorial(int n)* method will work:

- Base case: if *n* is negative or 0, the method returns 1
- General case: if *n* is greater than 0, the method returns *n* * *factorial*(*n* 2 1)

At lines 10–11, we make the initial call to the *factorial* method and print the result. We simply compute the factorial of 5. We can modify the example to prompt the user for another value.

We coded the *factorial* method at lines 14–24. The *factorial* method takes an *int* parameter named *n*, and returns the factorial of *n* as an *int*. At line 20, we test if *n* is less than or equal to 0. If that is true, we have reached the base case, and the *factorial* method returns 1. If *n* is greater than 0, the code skips to line 23, where we have coded the general case. We make a recursive call to the *factorial* method with an argument of (*n* 2 1). The value returned by that recursive call is then multiplied by *n* and the result is returned.

Figure 12.3 shows the output of Example 12.2.

Figure 12.3 Output of Example 12.2

Description

We can verify that factorial(5) is 120. Indeed,

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Description

To illustrate how recursive method calls are processed and their values are returned, let's modify Example 12.2 to include some output statements inside the *factorial* method. In this way, we can trace the recursive calls.

The JVM manages method calls using a **stack**. When a method is called, the JVM places the method's arguments and the caller's return address in a **frame** on the stack in a last-in, first-out order. Each method invocation adds another frame to the top of the stack. When a method completes and returns to the caller, its frame is removed from the top of the stack and is discarded. To help us trace the processing of recursive calls, we will use the *StackWalker* class, introduced in Java 9, which allows us to display the current contents of the stack at each invocation of the recursive method.

Example 12.3 is similar to Example 12.2 except that we have added statements to trace these features.

Example 12.3 Tracing Recursive Calls of the *factorial* Method

```java
/* Tracing the calculation of the factorial
   of a number using recursion
   Anderson, Franceschi
*/
import java.util.List;
import java.util.stream.Collectors;

public class RecursiveFactorialWithStackTrace
{
  public static void main( String [ ] args )
  {
    // compute factorial of 5 and output it
    System.out.println( "\nFactorial( 5 ) is "
                      + factorial( 5 ) );
  }

  /** recursive factorial method
  *   @param   n  a positive integer
  *   @return   the factorial of n
  */
  public static int factorial( int n )
  {
    List<StackWalker.StackFrame> stack = StackWalker.getInstance( )
             .walk( s -> s.skip(1).collect( Collectors.toList( ) ) );

    System.out.println( "factorial( " + n + " ) called" );
    System.out.println( "\tCurrent Stack" );
    for ( StackWalker.StackFrame f : stack )
      System.out.println( "\t\t" + f.getLineNumber( ) + " "
                                 + f.getMethodName( ) );

    if ( n == 0 )  // base case
    {
      System.out.println( "\nBase case detected\n" );

      System.out.println( "factorial( " + n + " ) returning 1" );
      System.out.println( "\tCurrent Stack"  );
      for ( StackWalker.StackFrame f : stack )
        System.out.println( "\t\t" + f.getLineNumber( ) + " "
                                   + f.getMethodName( ) );

      return 1;
    }
    else        // general case
    {
      int factorialNMinus1 = factorial( n - 1 );

      System.out.println( "factorial( " + n + " ) returning "
                           + n + " * " + factorialNMinus1 );
      System.out.println( "\tCurrent Stack"  );
      for ( StackWalker.StackFrame f : stack )
        System.out.println( "\t\t" + f.getLineNumber( ) + " "
                                   + f.getMethodName( ) );

      return ( n * factorialNMinus1 );
    }
  }
}
```

Description

- each call to the factorial method and its argument
- the detection of the base case and the value returned at that point
- the expression that evaluates to the return value
- the current stack at each method invocation

The *factorial* method is coded at lines 17–57.

When the method begins executing, we capture the current state of the stack into a *List* of *StackWalker.StackFrame* elements at lines 23–24. We get the current stack by calling the *getInstance* method of the *StackWalker* class. We then call the *walk* method, also in the *StackWalker* class, which creates a **stream** of stack frames. Streams, introduced in Java 8, are sequences of elements to which aggregate functions can be applied. The *walk* method, using a lambda expression, then traverses the stream, calling the *Stream collect* method, which calls on the *Collectors' static toList* method to accumulate the stack frames into a list. We skip the first frame, (*skip(1)*), because the first frame represents the method call to create the list.

The *StackWalker* class is in *java.lang*, so we do not need an *import* statement. On lines 5–6, we import the *List* interface and the *Collectors* class from *java.util* and *java.util.stream*, respectively.

At line 26, we output a message indicating that the *factorial* method has been called, along with the parameter value. When the base case is detected, we output a message indicating that the method has reached the base case and report its return value (lines 34–36). For the general case, we output the expression that will be returned by the method (lines 48–49). At each of these places, we also output the current state of the stack, displaying for each stack frame the line number from which the method was called and the name of the method. On lines 27–30 we output the stack when the *factorial* method starts executing. On lines 37–40 we output the stack when the method has detected the base case and is returning a known value (1), and on lines 50–53 we output the stack after the method makes a recursive call in the general case.

Figure 12.4 shows the output of Example 12.3. We can see the recursive calls to the *factorial* method with the value of the argument being reduced by 1 until the base case, 0, is reached. At that point, each recursively called method, in turn, returns a value to its caller until the initial invocation of the method returns the value 120. We also see that each time we detect the general case, another invocation of the *factorial* method is added to the stack. Once the base case is reached, the *factorial* methods on the stack start returning their value, and as they complete, the frame for their method invocation is removed from the stack.

Figure 12.4 The Trace of the *factorial* Method (*continues*)

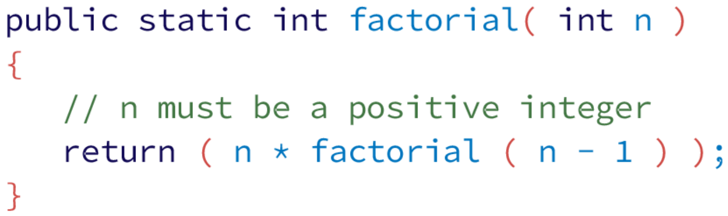
Description

Identifying the base case is critical. If a recursive method never reaches a base case, the method continues calling itself indefinitely, causing the JVM to continue placing frames on the stack until memory for the stack is full. At this time, the JVM generates a *StackOverflowError*, which terminates the program.

COMMON ERROR TRAP

Failure to code the base case will result in a run-time error.

For example, if we did not code the base case in our *factorial* method, the method would look like the following:



```
public static int factorial( int n )
{
   // n must be a positive integer
   return ( n * factorial ( n - 1 ) );
}
```

Description

When the method is called, the recursive calls keep being made because the base case is never reached. This eventually generates a *StackOverflowError*. **Figure 12.5** shows the first few lines of output from a run of Example 12.2 (the *RecursiveFactorial* class) with lines 20 to 22 commented out.

Figure 12.5 The First Few Lines of a Run of Example 12.2 If the Base Case Is Not Coded

Description

CODE IN ACTION

Within the online resources, you will find a movie with a step-by-step illustration of computing a factorial using recursion. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

12.2.2 Computing the Greatest Common Divisor

A common algebra problem is to calculate the greatest common divisor, or **gcd**, of two positive integers. The gcd is the greatest positive integer that divides evenly into both numbers.

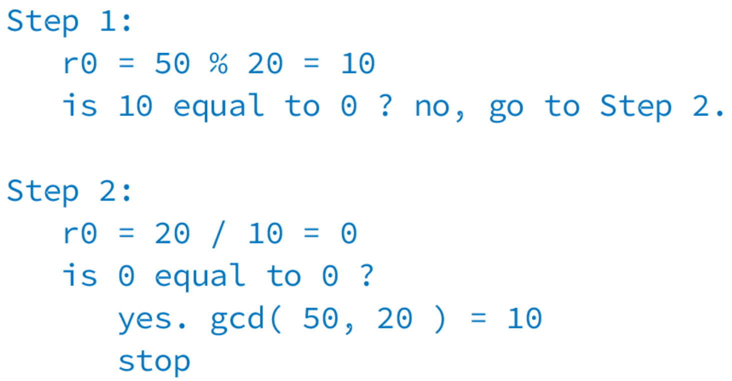
For example, consider 50 and 20. We can figure in our head that 5 divides evenly into both numbers, but so does 10. Since we can't find a number greater than 10 that divides evenly into both numbers, 10 is the gcd of 50 and 20.

It is easy to guess the gcd of two small numbers, but it is more difficult to guess the gcd of two large numbers, such as 474 and 162. The following Euclidian algorithm finds the gcd of two positive integers *a* and *b*. This algorithm derives from the fact that the gcd of two integers *a* and *b* (with *a* > *b*) is the same as the gcd of *b* and the remainder of *a* / *b*.

```
Step 1:
  r0 = a % b
  if ( r0 is equal to 0 )
      gcd ( a, b ) = b
      stop
  else
      go to step 2

Step 2:
  repeat step 1 with b and r0, instead of a and b.
```

Description

Let's run the algorithm on our first example, 50 and 20. We substitute 50 for *a* and 20 for *b*.



```
Step 1:
    r0 = 50 % 20 = 10
    is 10 equal to 0 ? no, go to Step 2.

Step 2:
    r0 = 20 / 10 = 0
    is 0 equal to 0 ?
        yes. gcd( 50, 20 ) = 10
        stop
```

Description

Therefore, the gcd of 50 and 20 is 10.

Let's now run the algorithm on our second example, 474 and 162.

The remainder of 474 divided by 162 is 150
150 is not equal to 0
so we take the remainder of 162 divided by 150, which is 12
12 is not equal to 0
so we take the remainder of 150 divided by 12, which is 6
6 is not equal to 0
so we take the remainder of 12 divided by 6, which is 0
0 is equal to 0
so the gcd of 474 and 162 is 6

Let's go back to our algorithm and look at Step 1 as a method taking two parameters, *a* and *b*. Step 2 is a method call to Step 1 with two different parameters, *b* and *r*0. It is very simple to calculate *r*0, since *r0* is the remainder of the division of *a* by *b*. Using the modulus operator, *r*0 is *a % b*. Therefore, this algorithm can easily be coded as a recursive method.

Let's call the two parameters of the method, *dividend* and *divisor*, in that order.

When the remainder of the division of *dividend* by *divisor* is 0, we have reached the base case and the method returns *divisor*. The general case is when the remainder of the division of *dividend* by *divisor* is not 0. The method then calls itself with *divisor* and the remainder of the division of *dividend* by *divisor*.

Example 12.4 shows the code for the recursive implementation of the greatest common divisor solution.

Example 12.4 Computing the GCD of Two Integers Using Recursion

```
 1 /* Computing the greatest common divisor using recursion
 2    Anderson, Franceschi
 3 */
 4
 5 public class RecursiveGCD
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     // compute and output gcd of 474 and 162
10     System.out.println( "The GCD of " + 474 + " and "
11                          + 162 + " is " + gcd( 474, 162 ) );
12   }
13
14   /** recursive gcd method
15   *   @param    dividend  the first strictly positive integer
16   *   @param    divisor   the second strictly positive integer
17   *   @return             the gcd of dividend and divisor
18   */
19   public static int gcd( int dividend, int divisor )
20   {
21     if ( dividend % divisor == 0 )     // base case
22       return divisor;
23     else                               // general case
24       return ( gcd ( divisor, dividend % divisor ) );
25   }
26 }
```

Description

We make the call to the *gcd* method at lines 10–11 with arguments 474 and 162 and output the result.

The *gcd* method is coded from lines 14–25. The method header shows that the *gcd* method takes two *int* parameters named *dividend* and *divisor*, and returns an *int*, the greatest common divisor of *dividend* and *divisor*. At line 21, we check for the base case by testing if the remainder of the integer division of *dividend* by *divisor* is 0. If so, the *gcd* method returns *divisor* without making another recursive call.

If the remainder is not 0, we are in the general case, so we make a recursive call at line 24 with the arguments *divisor* and the remainder of the division (*dividend % divisor*). We return the value returned by that call.

Figure 12.6 shows the output of Example 12.4.

Figure 12.6 Output of Example 12.4

Description

As we did with the recursive *factorial* method, let's modify Example 12.4 to include some output statements inside the *gcd* method in order to trace the recursive calls.

The *gcd* method in **Example 12.5** (lines 16–60) is the same as the *gcd* method in Example 12.4, except that each time the method is called, we print the parameter values and result of the modulus operation (lines 27–29) to verify that the method is correctly detecting the general and base cases. We also print a message when the base case is reached (lines 37–38). At lines 49–51, we output the value returned by the method in the general case. As in Example 12.3, at each of these places, we also output the current state of the stack (lines 30–33, 39–42, and 53–56), displaying for each stack frame the line number from which the method was called and the name of the method.

Example 12.5 Tracing the Recursive Calls of the *gcd* Method

```java
/* Computing the greatest common divisor using recursion
   Anderson, Franceschi
*/
import java.util.List;
import java.util.stream.Collectors;

public class RecursiveGCDWithStackTrace
{
  public static void main( String [ ] args )
  {
    // compute gcd of 474 and 162 and output it
    System.out.println( "\nThe GCD of " + 474 + " and "
                         + 162 + " is " + gcd( 474, 162 ) );
  }

  /** recursive gcd method with trace
   *   @param    dividend   the first strictly positive integer
   *   @param    divisor    the second strictly positive integer
   *   @return              the gcd of dividend and divisor
  */
  public static int gcd( int dividend, int divisor )
  {
    List<StackWalker.StackFrame> stack = StackWalker.getInstance( )
           .walk( s -> s.skip(1).collect( Collectors.toList( ) ) );


    System.out.print( "gcd( " + dividend + ", " + divisor + " )" );
    System.out.println( "   " + dividend + " % " + divisor + " = "
                         + ( dividend % divisor ) );
    System.out.println( "\tCurrent stack" );
    for ( StackWalker.StackFrame f : stack )
      System.out.println( "\t\t" + f.getLineNumber( )
                            + " " + f.getMethodName( ) );

    if ( dividend % divisor == 0 )  // base case
    {
      System.out.println( "\nbase case reached, returning "
                           + divisor );
      System.out.println( "\tCurrent stack" );
      for ( StackWalker.StackFrame f : stack )
        System.out.println( "\t\t" + f.getLineNumber( )
                            + " " + f.getMethodName( ) );

      return divisor;
    }
    else                           // general case
    {
      int temp = gcd( divisor, dividend % divisor );
      System.out.println( "gcd( " + divisor + ", "
                          + ( dividend % divisor )
                          + " ) returning " + temp );

      System.out.println( "\tCurrent stack" );
      for ( StackWalker.StackFrame f : stack )
        System.out.println( "\t\t" + f.getLineNumber( )
                            + " " + f.getMethodName( ) );

      return temp;
    }
  }
}
```

Description

Figure 12.7 shows the output of Example 12.5. We can see the recursive calls all the way to the base case, and the return value from each recursive call. We also see that each time we detect the general case, we call the *gcd* method recursively, and the current method is added to the stack because the current method cannot complete until the invoked method returns. Once the base case is reached, the *gcd* methods on the stack start returning their value. One by one each method completes and its stack frame is removed from the stack. Note that the return value stays the same throughout the process. Such a recursive method is called **tail recursive** because the method does no further processing when the recursive call returns.

Figure 12.7 The Trace of the *gcd* Method

Description

CODE IN ACTION

Within the online resources, you will find a movie with a step-by-step illustration of computing a GCD and various other recursive methods. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

```
C( 5, 2 ) = 10
```

12.3 Recursion with Two Base Cases

Recursive formulations can be more complex than the examples we have discussed. The general case can involve more than one recursive call, with different arguments. This, in turn, means that we can have more than one base case.

Suppose we are playing a networked video game online. There are *n* players who would like to play. Unfortunately, that game can be played with only *p* players. We will make the assumption that *p* is an integer between 0 and *n* (for instance, *n* could be 100 and *p* could be 8). Otherwise, we simply cannot play the game.

Our problem is to determine how many different ways we can choose *p* players from among *n* players. We will call that number *Combinations(n, p)*.

The math formula for *Combinations*(*n, p*) is:

$$Combinations(n, p) = n!/((n - p)! * p!)$$

Our goal here is to come up with a recursive solution to the problem and thus to code *Combinations*(*n, p*) recursively.

There are some obvious cases to consider. If we have the same number of players as the number who can play the game, then *p* equals *n*, and we pick all the players. There is only one way to do that, so *Combinations*(*n, n*) = 1.

If the game requires no players, then *p* equals 0, and we do not pick any players. Again, there is only one way to do that, so *Combinations*(*n*, 0) = 1.

But what is the answer in the general case where the value of *Combinations*(*n, p*) may not be so obvious?

One way to look at that problem is as follows:

Among these *n* potential players, let’s focus on one player in particular. We will call that player Louis. We can either pick Louis or not pick Louis. Therefore, the total number of possibilities of picking *p* players among *n* potential players is equal to the sum of the following two numbers:

- the number of possibilites of picking *p* players, including picking Louis, among *n*
- the number of possibilities of picking *p* players, without picking Louis, among *n*

If we pick Louis, then we will have to choose (*p* 2 1) more players. But we cannot pick Louis again, so there are only (*n* 2 1) potential players left. The number of such possibilities is *Combinations(n 2 1, p 2 1)*.

If we do not pick Louis, then we still have to choose *p* players. But since we are not picking Louis, there are only (*n* 2 1) potential players left. The number of such possibilities is *Combinations(n 2* 1*, p)*.

Therefore, we can write the following recursive formula:

```
Combinations( n, p ) = Combinations( n - 1, p - 1 )
                       + Combinations( n - 1, p )
```

Description

If we look at the two terms on the right side of the preceding formula, we can see that:

- In the first term, both parameters, *n* and *p*, have been decreased by 1.
- In the second term, one parameter, *n*, has been decreased by 1, while *p* is unchanged.

Therefore, solving the problem of computing *Combinations*(*n, p*) using this formula translates into solving two similar, but smaller, problems. That is our general case.

Our next concern is to decide what the base case or cases are. In other words, as we apply the preceding formula repeatedly, when will we reach an easy-to-solve problem? Since we have two recursive terms on the right side of the formula, we will have two base cases.

Let's look at the first term, *Combinations*(*n – 1, p – 1*). We can see that both *n* and *p* decrease by 1 at the same time. When we start, *p* is greater than or equal to 0 and less than or equal to *n*. Therefore, as we keep applying the formula and concentrate on the first term, we can see that *p* will eventually reach 0, and that *p* will reach 0 before *n* does. As discussed earlier, *Combinations*(*n*, 0) = 1, because there is only one way to pick 0 players from a set of *n* players—do not pick any. This is one base case.

Let's now look at the second term, *Combinations*(*n* – 1, *p*). We can see that *n* decreases by 1 while *p* is unchanged. We know that *p* must be less than or equal to *n* (we cannot pick more than *n* players among *n* players). As *n* decreases and *p* does not, *n* will eventually reach *p*. As discussed earlier, *Combinations*(*n, n*) = 1, because there is only one way to pick *n* players among *n* players—pick them all. This is our other base case.

Example 12.6 shows the code for this example.

Example 12.6 Computing Combinations Recursively

```
/* Computing the number of combinations
   of picking p objects among n, using recursion
   Anderson, Franceschi
*/

public class RecursiveCombinations
{
  public static void main( String [ ] args )
  {
    // compute and output number of combinations
    System.out.println( "C( 5, 2 ) = "
                         + combinations( 5, 2 ) );
  }

  /** recursive combinations method
  *   @param    n a positive number
  *   @param    p a positive number, less than or equal to n
  *   @return   the number of combinations of choosing p among n
  */
  public static int combinations( int n, int p )
  {
    if ( p == 0 )           // base case # 1
      return 1;
    else if ( n == p )      // base case # 2
      return 1;
    else                    // general case
      return ( combinations( n - 1, p - 1 )
              + combinations( n - 1, p ) );
  }
}
```

Description

In this example, we use the *combinations* method to compute the number of ways of picking 2 players from among 5.

We call the *combinations* method with arguments, 5 and 2, and output the returned value at lines 11–12.

The *combinations* method is coded at lines 15–29. The method header, at line 20, shows that the *combinations* method takes two *int* parameters, the number of players (*n*) and the number of players to select (*p*). The return value, an *int*, is the number of combinations of picking *p* players among *n*.

At line 22, we test for the first base case (*p* == 0). If true, we return 1. If *p* is not equal to 0, we test for the second base case (*n* is equal to *p*). If that is *true*, we return 1. If *p* is not equal to 0 and *n* is not equal to *p*, then we are in the general case and the code skips to lines 27–28. We make two recursive calls to the *combinations* method. The first recursive call is with arguments *n* – 1 and *p* – 1. The second recursive call is with arguments *n* – 1 and *p*. We add the values returned by these two recursive calls and return the result.

The output of Example 12.6 is shown in **Figure 12.8**.

Figure 12.8 Output of Example 12.6

Description

We can verify that our algorithm is correct. As discussed earlier,

$$Combinations(n,p) = n!/\,((\,n-p\,)!*p!)$$

Description

Thus,

$$Combinations(\,5,2\,) = 5!/\,(\,3!*2!) = 10$$

Description

Those of us with a mathematics background can verify that

```
Combinations( n, p ) = Combinations( n - 1, p - 1 ) +
                       Combinations( n - 1, p )
```

Description

that is,

```
n! / ( ( n - p )! * p! ) = ( n - 1 )! / ( ( n - p )! * ( p - 1 )! )
                          + ( n - 1 )! / ( ( n - 1 - p )! * p! )
```

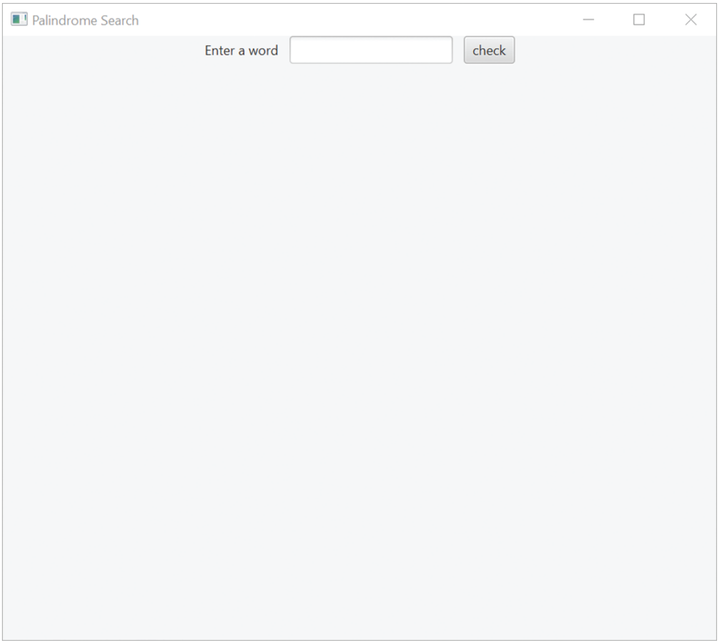
Description

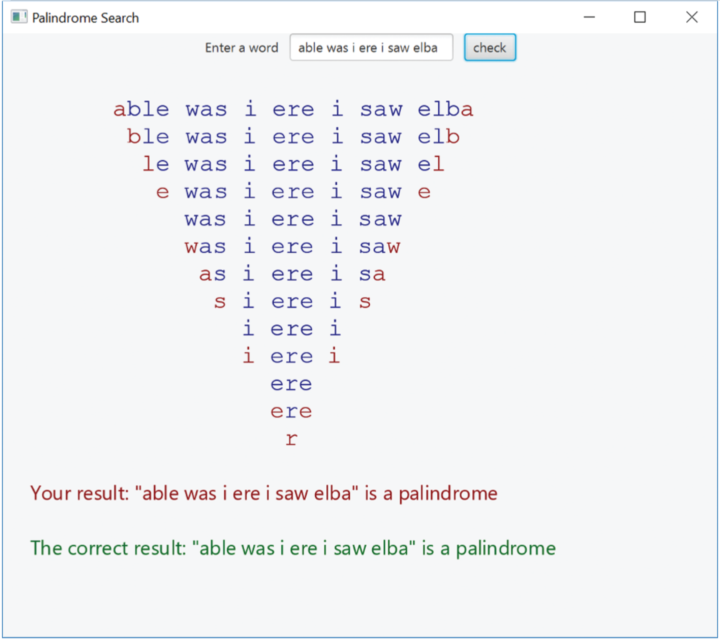
What happens if we code for only one base case when there are two or more base cases?

COMMON ERROR TRAP

There can be more than one base case. Failing to take into account all base cases can result in a *StackOverflowError* at run time.

When the method is called, the recursive calls will continue to be made, because the missing base cases will never be detected. This will eventually generate a *StackOverflowError*.


Palindrome Search
Enter a word
check

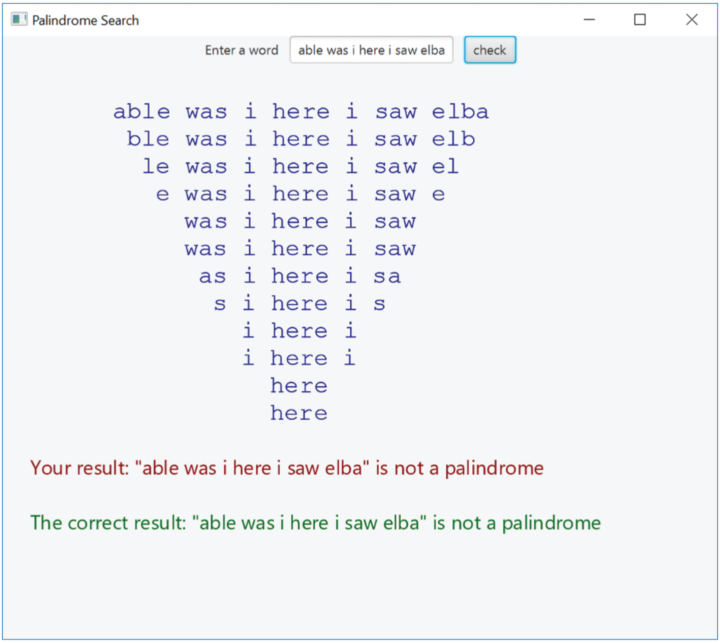

Palindrome Search
Enter a word
check

able was i ere i saw elba

```
able was i ere i saw elba
 ble was i ere i saw elb
  le was i ere i saw el
   e was i ere i saw e
     was i ere i saw
     was i ere i saw
      as i ere i sa
       s i ere i s
         i ere i
         i ere i
           ere
           ere
            r
```

Your result: "able was i ere i saw elba" is a palindrome

The correct result: "able was i ere i saw elba" is a palindrome



```
able was i here i saw elba
 ble was i here i saw elb
  le was i here i saw el
   e was i here i saw e
     was i here i saw
     was i here i saw
      as i here i sa
       s i here i s
         i here i
         i here i
           here
           here
```

Your result: "able was i here i saw elba" is not a palindrome

The correct result: "able was i here i saw elba" is not a palindrome

12.4 Programming Activity 1: Checking for a Palindrome

In this activity, you will work with recursion to perform this function:

Code a recursive method to determine if a *String* is a palindrome.

A palindrome is a word, phrase, or sentence that is symmetrical; that is, it is spelled the same forward and backward. Examples are “otto,” “mom,” “madam,” and “able was I ere I saw elba.”

How can we determine, using recursion, whether a *String* is a palindrome?

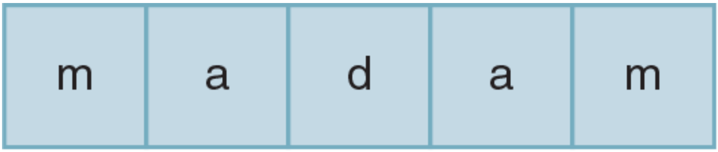
If the *String* has two or more characters, we can check if the first and last characters are identical. If they are not identical, then the *String* is not a palindrome. That is a base case.

If the first and last characters are identical, then we need to check if the substring comprised of all the characters between the first and last characters is a palindrome. That is the general case.

If the *String* is a palindrome, each recursive call will reduce the size of the argument—that is, the number of characters in the argument *String*—by 2. Eventually, the recursive calls will result in a *String* argument consisting of 0 or 1 character. Both are trivial palindromes. That is our second base case. Note that we will reach this base case only if the *String* is a palindrome. Indeed, if the *String* is not a palindrome, the recursive calls will detect the first base case as soon as the first and last characters of the *String* argument are different, and the recursive method will return *false*.

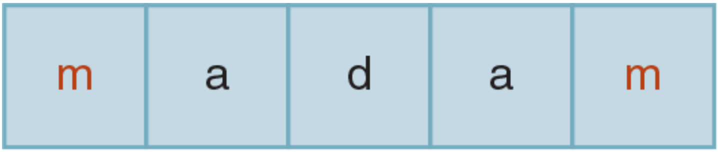
For example, to check if “madam” is a palindrome, we take the following steps.

Here is the original *String*:



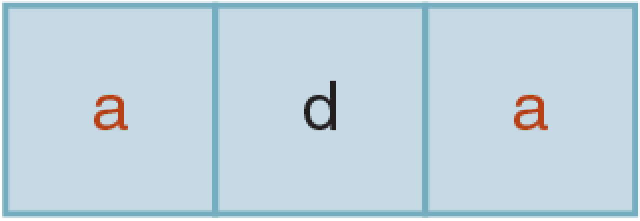
m	a	d	a	m

We compare the first and last characters.



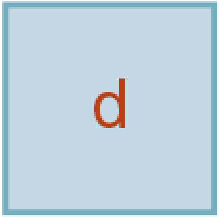
m	a	d	a	m

They are equal, so we now check the substring comprised of the characters between the first and last characters. Again, we compare the first and last characters of this substring.



a	d	a

They are equal, so we now check the substring comprised of the characters between the first and last characters.

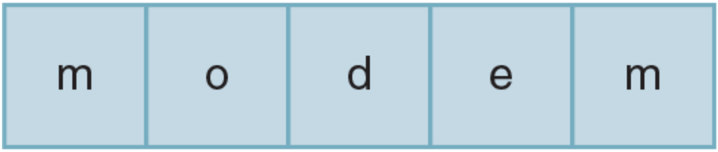


d

There is only one character in this substring, so we have reached our second base case. The *String* “madam” is a palindrome.

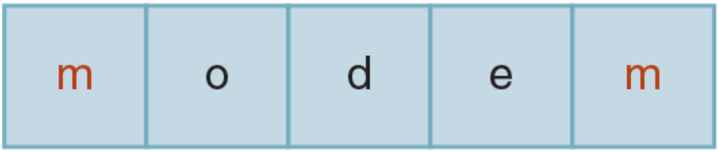
Let’s now check if “modem” is a palindrome.

Here is the original *String*:



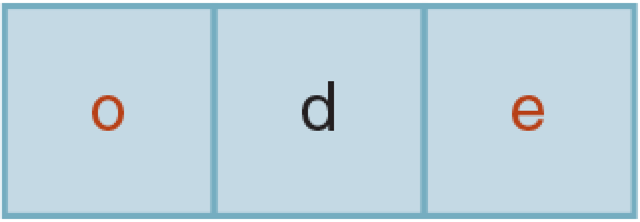
m	o	d	e	m

We compare the first and last characters.



m	o	d	e	m

They are equal, so we now check the substring comprised of the characters between the first and last characters. Again, we check the first and last characters.



o	d	e

They are not equal, so we have reached the first base case. The *String* “modem” is not a palindrome.

Instructions

In this chapter's Programming Activity 1 folder, you will find the source files needed to complete this activity. Copy all of the files to a folder on your computer. Note that all files should be in the same folder.

Open the *PalindromeDrawing.java* source file. Searching for five asterisks (*****) in the source code will position you to the location where you will add your code. In this task, you will fill in the code inside the *recursivePalindrome* method to determine if a *String* representing a word or a sentence is a palindrome. The method returns *true* if the *String* is a palindrome, *false* if the *String* is not a palindrome. **Example 12.7** shows the section of the *PalindromeDrawing* source code where you will add your code.

Example 12.7 Location of Student Code in *PalindromeDrawing*

```
public boolean recursivePalindrome( String pal )
{
  // ***** Student writes the body of this method *****

  // Using recursion, determine if a String representing
  // a word or a sentence is a palindrome
  // If it is, return true, otherwise return false

  // We call the animate method inside the body of this method
  // The call to animate is already coded below

  animate( pal );

  //
  // Student code starts here
  //

  return true; // replace this dummy return statement

  //
  // End of student code - PA 1
  //
}
```

Description

The framework will animate your code so that you get some feedback on the correctness of your code. It will display the argument *String* passed to the recursive method at each recursive call of that method. Your result will be displayed in red and the correct result will be displayed in green.

To test your code, compile *PalindromeDrawing.java* and run *PalindromeApplication*. When the program begins, you will see a text box in which to enter a word or a sentence, as shown in **Figure 12.9**.

Figure 12.9 Opening Window

Description

Click “check” to animate your code.

If you enter an empty *String* or a *String* with more than 26 characters, the text box will display “enter 1–26 characters”. This part is already coded for you.

Figure 12.10 shows the output if you enter “able was I ere I saw elba.” We can see the argument *String* of our recursive method shrinking by two characters at each recursive call until we reach a base case.

Figure 12.10 Sample Final Screen for Programming Activity 1 When a Palindrome Is Found

Description

If you insert an extra “h” into the preceding phrase, and enter “able was I here I saw elba,” which is not a palindrome, the final result of your animation is shown in **Figure 12.11**. When the argument *String* of our recursive method becomes “here,” the recursive calls stop and the method returns *false*.

Figure 12.11 Sample Final Screen for Programming Activity 1 When a Palindrome Is Not Found

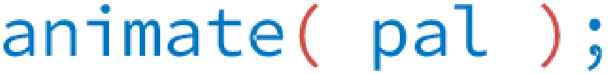
Description

Task Instructions

Inside the method *recursivePalindrome* of class *PalindromeDrawing* write the code to solve the palindrome problem:

- The *recursivePalindrome* method header has already been coded for you. Write the code to check if the parameter of the method, *pal*, is a palindrome. Return *true* if it is, *false* if it is not. Your method should be recursive; that is, it should call itself. We have provided a dummy *return* statement so that the code will compile. You should replace the dummy statement with your appropriate *return* statement.

- Be sure your code ignores case differences; that is, Otto and Racecar are indeed palindromes.
- The call to the *animate* method has already been written for you. It should be the first statement in the body of the method and is simply:



```
animate( pal );
```

Troubleshooting

If your method implementation does not animate or animates incorrectly, check these items:

- Check the feedback on the output to see if your code gives the correct result.
- Verify that you coded the base cases correctly.
- Verify that you coded the general case and its corresponding recursive call correctly.

DISCUSSION QUESTIONS

1. What are the base cases for this method?
2. Is this method tail recursive?
3. What happens if you do not code one of the base cases?

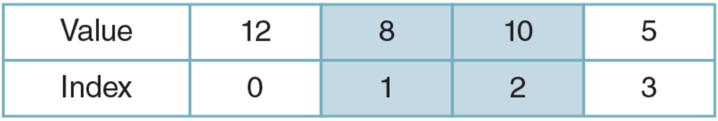
```
total = 35
```

```
sum called with index 0
sum called with index 1
sum called with index 2
sum called with index 3
sum called with index 4
total = 35
```

12.5 Simple Recursion on an Array

When coding a recursive method that performs an aggregate operation on an array, we typically define that method as performing that operation on a subarray.

We can define a subarray of an array with two integers: a starting index and an ending index for that subarray. For example, a starting index of *1* and an ending of *2* define a subarray of two elements as shown here:

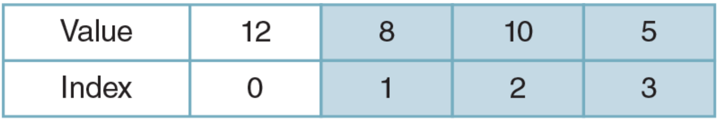


Value	12	8	10	5
Index	0	1	2	3

Description

Above, a starting index of *0* and an ending index of *3* would define a subarray that is the whole array.

In many problems, it is enough to define the subarray with just a starting index; in this case, the implicit ending index is the last index of the array. For example, a starting index of *1* defines a subarray of three elements as shown here:



Value	12	8	10	5
Index	0	1	2	3

Description

Above, a starting index of *0* would define a subarray that is the whole array.

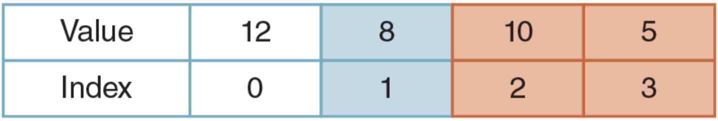
Thus, we can define the general case as follows:

1. Process the first element of the subarray
2. Process the remaining subarray

For a first element to exist, the subarray must not be empty; thus, the base case is when the subarray is empty. That happens when the starting index is equal to the length of the array; that is, it is *1* beyond the last index of the array.

For example, if we want to compute the sum of the elements of a subarray, we can do it as follows:

Sum = first element + sumof the elements of the remaining subarray



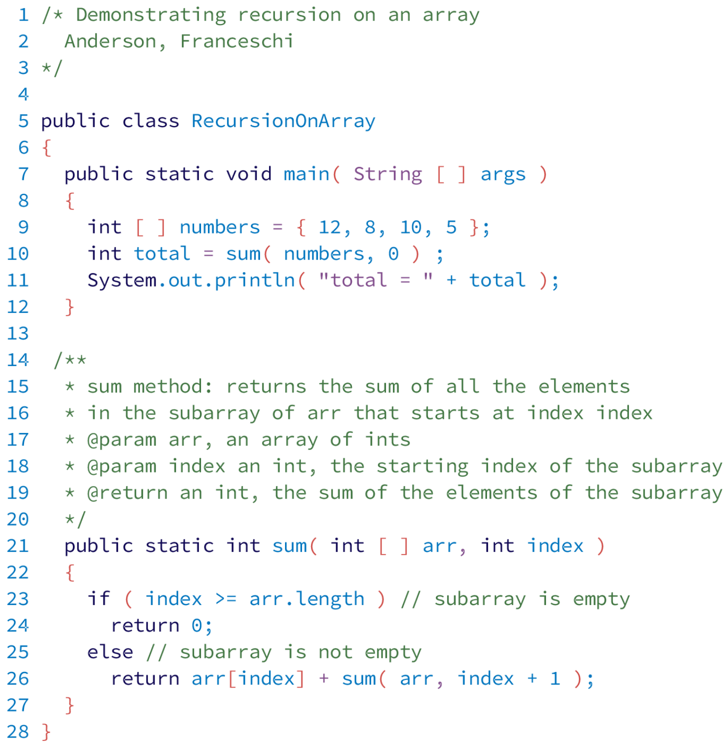
Value	12	8	10	5
Index	0	1	2	3

Description

In the array above, the sum of the elements of the subarray starting at index 1 is equal to element at index 1 (8) plus the sum of the elements of the subarray starting at index 2 (10 + 5). Eventually, the remaining subarray is empty and the sum of the elements of that empty subarray is *0*.

Example 12.8 shows a recursive method computing the sum of the elements of an array of integers (lines 14–27). The *index* parameter (line 21) represents the starting index of the subarray of *arr*, the other parameter of *sum*, that the *sum* method operates on. At line 23, we test if the subarray is empty and return *0* if it is. If the subarray is not empty, we return the sum of its first element and the sum of the remaining subarray at line 26. In the *main* method, at line 10, we call *sum* on the whole array *numbers*, that is, the subarray that starts at index *0*. The output is shown in **Figure 12.12**.

Example 12.8 Summing the Elements of an Array Using Recursion

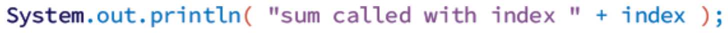


```
 1 /* Demonstrating recursion on an array
 2    Anderson, Franceschi
 3 */
 4
 5 public class RecursionOnArray
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     int [ ] numbers = { 12, 8, 10, 5 };
10     int total = sum( numbers, 0 ) ;
11     System.out.println( "total = " + total );
12   }
13
14  /**
15   * sum method: returns the sum of all the elements
16   * in the subarray of arr that starts at index index
17   * @param arr, an array of ints
18   * @param index an int, the starting index of the subarray
19   * @return an int, the sum of the elements of the subarray
20   */
21   public static int sum( int [ ] arr, int index )
22   {
23     if ( index >= arr.length ) // subarray is empty
24       return 0;
25     else // subarray is not empty
26       return arr[index] + sum( arr, index + 1 );
27   }
28 }
```

Description

Figure 12.12 Output of Example 12.8

If we want to trace all the recursive calls to *sum*, we can add the following statements before line 23:



```
System.out.println( "sum called with index " + index );
```

Description

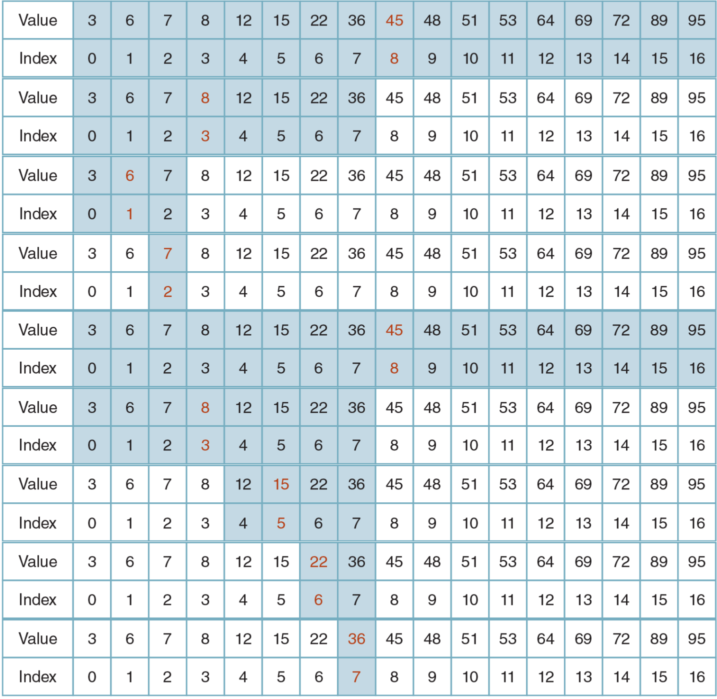
Figure 12.13 shows the resulting output when we make the original call: *sum(numbers, 0)*. The original call and the last call are highlighted.

Figure 12.13 Output of Example 12.8 with Tracing Added

Description

```
Enter a value to search for > 7
7 found at index 2
```

```
Enter a value to search for > 34
34 not found
```



Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

12.6 Binary Search: A Recursive Solution

We can use a binary search algorithm to search a sorted array for a given value.

Let’s look at how we can define a recursive solution to this problem. We will assume that the array is sorted in ascending order.

Again, we need to define the base cases and the general case, and the general case must reduce the size of the problem.

When searching for a value in an array, we have two possible outcomes:

- We find the value and return its array index.
- We do not find the value and return –1.

Overall, our strategy is this. First, we will look at the middle element of the array. If the value of the middle element is the value we are looking for, we will return its index. That is our first base case.

If the value of the middle element is greater than the value we are looking for, then the value we are looking for cannot be found in elements with array indexes higher than the index of the middle element. Therefore, we will continue our search in the lower half of the array only. We will do that by making a recursive call to our search method, specifying the lower half of the original array as the subarray to search.

Similarly, if the value of the middle element is lower than the value we are looking for, then the value we are looking for cannot be found in elements with array indexes lower than the index of the middle element. Therefore, we will continue our search in the upper half of the array only. We will do that by making a recursive call to our search method, specifying the upper half of the original array as the subarray to search. That is our formulation for the general case.

As we continue searching, the size of the subarray that we search will shrink with every recursive call. Indeed, every recursive call cuts the size of the subarray we search in half. In this recursive algorithm, not only does the size of the problem decrease with each recursive call, but it also decreases by a large amount.

If the value we are looking for is not in the array, the part of the array that we are searching will continue shrinking until it is empty. At that point, we know that we will not find our value in the array. We have reached our other base case, and we return –1.

Example 12.9 shows the code for a recursive binary search.

Example 12.9 Searching an Array Sorted in Ascending Order

```
 1 /* Searching a sorted array using recursion
 2    Anderson, Franceschi
 3 */
 4
 5 import java.util.Scanner;
 6
 7 public class RecursiveBinarySearch
 8 {
 9   public static void main( String [ ] args )
10   {
11     // define an array sorted in ascending order
12     int [ ] numbers = { 3, 6, 7, 8, 12, 15, 22, 36, 45,
13                         48, 51, 53, 64, 69, 72, 89, 95 };
14
15     Scanner scan = new Scanner( System.in );
16     System.out.print( "Enter a value to search for > " );
17     int value = scan.nextInt( );
18
19     int index = recursiveBinarySearch
20                 ( numbers, value, 0, numbers.length - 1 );
21     if ( index != -1 )
22       System.out.println( value + " found at index " + index );
23     else
24       System.out.println( value + " not found" );
25   }
26
27   /** recursiveBinarySearch method
28   *   @param    arr   the array sorted in ascending order
29   *   @param    key   the value to search for in the subarray
30   *   @param    start the subarray's first index
31   *   @param    end   the subarray's last index
32   *   @return   the array index at which key was found,
33   *             or -1 if key was not found
34   */
35   public static int recursiveBinarySearch
36                ( int [ ] arr, int key, int start, int end )
37   {
38     if ( start <= end )
39     {
40       // look at the middle element of the subarray
41       int middle = ( start + end ) / 2;
42
43       if ( arr[middle] == key )      // found key, base case
44         return middle;
45       else if ( arr[middle] > key )  // look lower
46         return recursiveBinarySearch( arr, key, start, middle - 1 );
47       else                           // look higher
48         return recursiveBinarySearch( arr, key, middle + 1, end );
49     }
50     else                             // key not found, base case
51       return -1;
52   }
53 }
```

Description

We coded the *recursiveBinarySearch* method at lines 27–52. That method takes four parameters: *arr*, the array we are searching; *key*, the value we are searching for; and *start* and *end*, which represent, respectively, the first and last index of the subarray of *arr* that we should search.

At line 38, we test if the subarray we are searching contains at least one element. If it does not, we have reached a base case and we know that we will not find *key*. Thus, we return −1 in the *else* clause at line 51. If the subarray has at least one element, we assign the index of the middle element of the subarray to *middle* at line 41. We then compare the array element at index *middle* to *key* at line 43. If they are equal, we have reached the other base case (we have found *key*) so we return *middle* at line 44.

If the array element at index *middle* is greater than *key*, we call the *recursiveBinarySearch* method with the subarray consisting of all elements with values lower than *middle* (from *start* to *middle* − 1) at line 46. If the array element at index *middle* is smaller than *key*, then we call the *recursiveBinarySearch* method with the subarray consisting of all elements with values higher than *middle* (from *middle* + 1 to *end*) at line 48. In both cases, whatever is returned by the recursive call is returned by the method.

In *main*, we begin by instantiating our array to search. Note that the values are in ascending order (lines 12–13). We then prompt the user for the search key and make the call to the recursive binary search method, passing the entire array as the subarray to search (lines 19–20). We output the result of our search at lines 21–24.

Figure 12.14 shows the output from Example 12.9 when the key value is found, and when the key value is not found.

Figure 12.14 Two Runs of Example 12.9

Description

Let’s run the preceding example on the value 7 in order to illustrate the various recursive calls and the case where the value is found.

Here is the array *numbers*, sorted in ascending order:

Description

We calculate the index *middle* by adding the indexes *start* and *end*, then dividing by 2. Thus, when the *recursiveBinarySearch* method is first called, *middle* is 8.

The element at index 8 (45) is greater than 7, so we call the *recursiveBinarySearch* method, searching the left subarray, highlighted here.

Description

The index *middle* is now calculated to be 3 ((0 + 7)/2).

The element at index 3 (8) is greater than 7, so we call the *recursiveBinarySearch* method, searching the left subarray, highlighted here.

Description

The index *middle* is now calculated to be 1 ((0 + 2)/2).

The element at index 1 (6) is smaller than 7, so we call the *recursiveBinarySearch* method, searching the right subarray, highlighted here.

Description

The index *middle* is now calculated to be 2 ((2 + 2)/2).

The element at index 2 (7), is equal to 7. We have found the value and return its index, 2.

Let’s now run the preceding example on the value 34 in order to illustrate the various recursive calls and the base case when the value is not found.

Here is the array *numbers* again:

Description

The index *middle* when the *recursiveBinarySearch* method is first called is 8 ((0 + 16)/2).

The element at index 8 (45) is greater than 34, so we call the *recursiveBinarySearch* method, searching the left subarray highlighted here.

Description

The index *middle* is now calculated to be 3 ((0 + 7)/2).

The element at index 3 (8) is smaller than 34, so we call the *recursiveBinarySearch* method, searching the right subarray, highlighted here.

Description

The index *middle* is now calculated to be 5 ((4 + 7)/2).

The element at index 5 (15) is smaller than 34, so we call the *recursiveBinarySearch* method, searching the right subarray, highlighted here.

Description

The index *middle* is now calculated to be 6 ((6 + 7)/2).

The element at index 6 (22) is smaller than 34, so we call the *recursiveBinarySearch* method, searching the right subarray, highlighted here.

Description

The index *middle* is now calculated to be 7 ((7 + 7)/2).

The element at index 7 (36) is larger than 34, so we call the *recursiveBinarySearch* method, searching the left subarray. However, that left subarray is empty. We have not found 34, so we return –1.

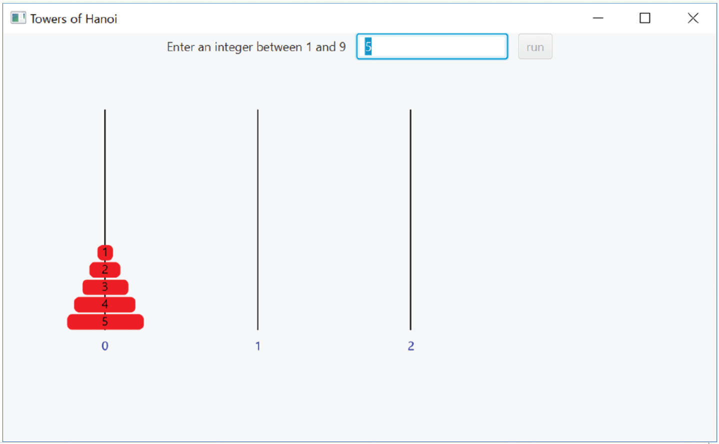
with these end-of-chapter questions

12.12.1	Multiple Choice Exercises Questions 2, 6, 7, 8
12.12.2	Reading and Understanding Code Questions 9, 10, 11, 12, 13, 14, 15, 20, 21, 22, 23
12.12.3	Fill In the Code Questions 24, 25, 26, 27, 28
12.12.4	Identifying Errors in Code Questions 29, 30, 31, 32
12.12.5	Debugging Area Questions 33, 34, 35, 36, 37, 38
12.12.6	Write a Short Program

Questions 42, 43, 45, 46, 47

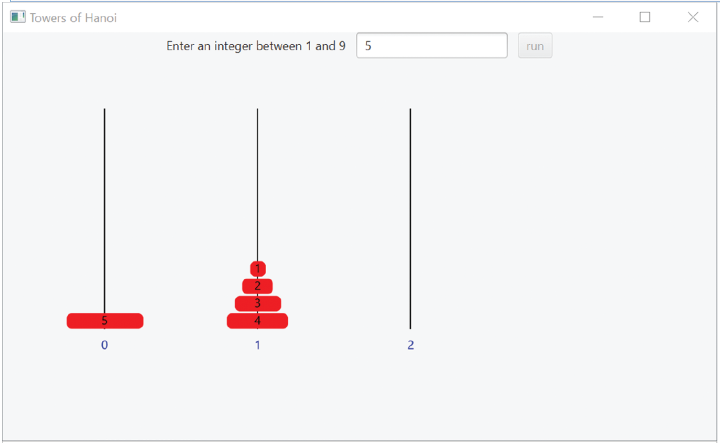
12.12.8 Technical Writing

Question 63


Towers of Hanoi

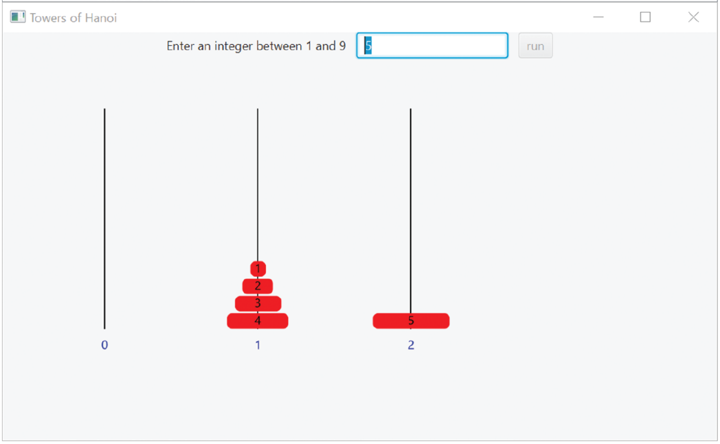
Enter an integer between 1 and 9 5

run
1
2
3
4
5
0
1
2


Towers of Hanoi

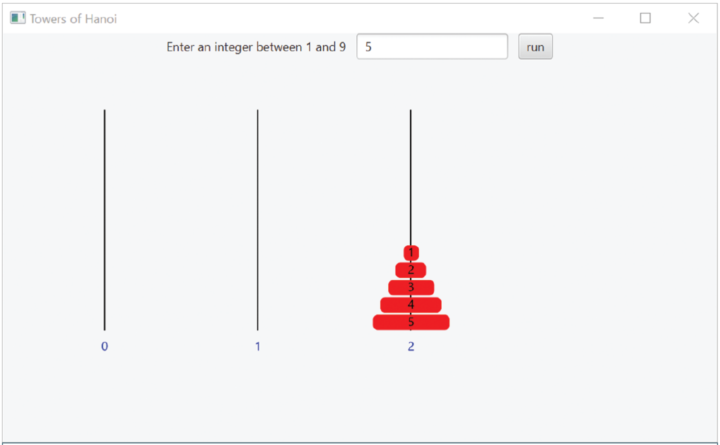
Enter an integer between 1 and 9 5

run
5
1
2
3
4
0
1
2

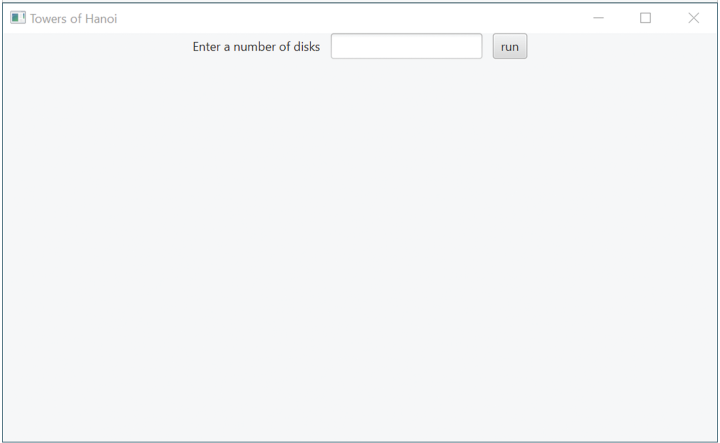

Towers of Hanoi

Enter an integer between 1 and 9 5

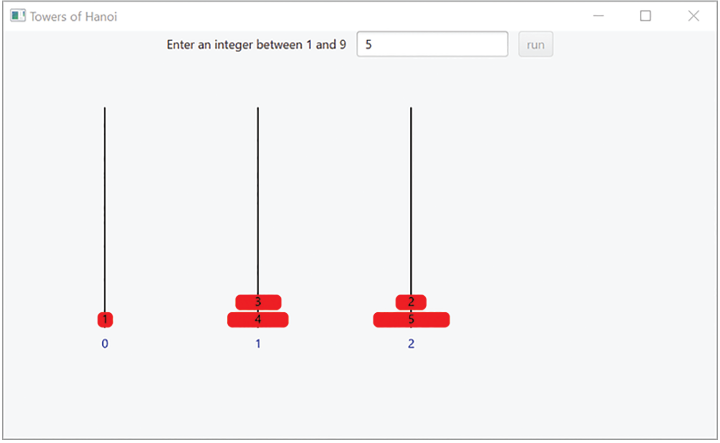
run
1
2
3
4
5
0
1
2


Towers of Hanoi
run
1
2
3
4
5
0
1
2

Enter an integer between 1 and 9 5


Towers of Hanoi
run

Enter a number of disks


Towers of Hanoi
run
1
3
4
2
5
0
1
2

Enter an integer between 1 and 9 5

12.7 Programming Activity 2: The Towers of Hanoi

A well-known problem that lends itself to an elegant recursive formulation is the Towers of Hanoi. Here it is:

- There are three towers, which we can represent as the source tower, the temporary tower, and the destination tower.
- We have a stack of *n* disks piled on the source tower; all the disks have a different diameter. The largest disk is at the bottom and the smallest disk is at the top.
- The goal is to transfer all the disks, one at a time, to the destination tower using all three towers for help. No larger disk can be placed on top of a smaller one.

The recursive solution to the problem for the general case (*n* >= 1) is as follows:

1. Transfer the top (*n* – 1) disks from the source tower to the temporary tower.
2. Transfer the one remaining disk (the largest) from the source tower to the destination tower.
3. Transfer the (*n* – 1) disks from the the temporary tower to the destination tower.

The base case, when *n 5* 0 (there are 0 disks to transfer), is to do nothing.

The first and third operations are simply recursive calls using a smaller number of disks (*n* – 1) than the original problem.

In the case of *n 5* 5, **Figures 12.15–12.18** illustrate the recursive solution and formulation. In the figures, the left, middle, and right towers represent the source, temporary, and destination towers, respectively.

Figure 12.15 Starting Position with Five Disks

Description

Figure 12.16 Position After Step 1

Description

Figure 12.17 Position After Step 2

Description

Figure 12.18 Position After Step 3

Description

In this activity, you will work with recursion to perform the following function:

Code a recursive method to solve the Towers of Hanoi problem

Instructions

In this chapter's Programming Activity 2 folder, you will find the source files needed to complete this activity. Copy all of the files to a folder on your computer. Note that all files should be in the same folder.

Open the *HanoiDrawing.java* source file. Searching for five asterisks (*****) in the source code will position you to the code section where you will add your code. In this task, you will fill in the code inside the *recursiveTOfH* method to solve the Towers of Hanoi problem. **Example 12.10** shows the section of the *HanoiDrawing* source code where you will add your code.

Example 12.10 Location of Student Code in *HanoiDrawing*

```
public void recursiveTOfH( int numDisks, int fromTower,
                           int toTower, int useTower )
{
   // ***** Student writes the body of this method *****
   //
   // Using recursion, transfer numDisks disks from the tower
   // fromTower to the tower toTower using the tower
   // useTower

   // The disks are numbered as follows: if we started with n disks,
   //   the disk at the top is disk # 1
   //   and the disk at the bottom is disk # n

   // We call the moveDisk method inside the body of this method

   // The moveDisk method moves one disk and takes 3 arguments:
   //   an int, representing the disk number to be moved
   //   an int, representing the tower to move the disk from
   //   an int, representing the tower to move the disk to
   // So if these three variables are:
   //   diskNumber, fromTower, and toTower
   //   then the call to moveDisks will be:

   //   moveDisk( diskNumber, fromTower, toTower );

   if ( numDisks > 0 )
   {
    // Student code starts here
    // 1. Move ( numDisks - 1 ) disks from fromTower
    //   to useTower using toTower
    // 2. Move one disk from fromTower to toTower
    //   Print a message to the screen, then
    //   call moveDisk in order to animate.

    // 3. Move ( numDisks - 1 ) disks from useTower to toTower
    //   using fromTower

   }

   // Base case: 0 disks to move ==> do nothing
   //
   // end of student code
   //
 }
```

Description

The framework will animate your code so that you get some feedback on the correctness of your code. It will display the disks being moved from one tower to another until the whole set of disks has been moved from the left tower to the right tower. Code to enforce the rules has already been written.

To test your code, compile *HanoiDrawing.java* and run *HanoiApplication*; when the program begins, a text box will allow you to enter the number of disks as shown in **Figure 12.19**.

Figure 12.19 Opening Window

Description

Click “run” to animate your code.

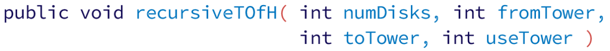
If you enter an integer less than 1 or greater than 9, the program will use a default value of 4. If you enter 5, as shown in Figure 12.19, the first screen will be as shown in Figure 12.15. **Figure 12.20** shows an intermediate position as you run your code.

Figure 12.20 An Intermediate Position in the Animation

Description

Task Instructions

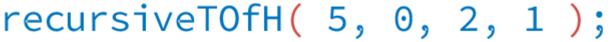
- In the file *HanoiDrawing.java* the *recursiveTOfH* method header is:



```
public void recursiveTOfH( int numDisks, int fromTower,
                           int toTower, int useTower )
```

Description

- This method takes four parameters: *numDisks*, representing the number of disks to be moved, and 3 *ints* representing the tower to move the disks from, the tower to move the disks to, and the tower to use to accomplish that task of moving *numDisks* disks from tower *fromTower* to tower *toTower*. For instance, with five disks, our method call in the *main* method is:



```
recursiveTOfH( 5, 0, 2, 1 );
```

Description

- The preceding method call is interpreted as: move 5 disks from tower 0 to tower 2, using tower 1 as a temporary holding tower.

- Your code goes in three places, all of them inside the *if* statement.

    1. First, you need to move all the disks except the bottom one from the *fromTower* (source tower, “left tower” on the figures) to the *useTower* (temporary tower, “middle tower” on the figures) using the *toTower* (“destination tower, right tower” on the figures). You do this by calling *recursiveTOfH* with the appropriate arguments.
    2. Then, you need to move the bottom disk from the *fromTower* (source tower, “left tower” on the figures) to the *toTower* (destination tower, “right tower” on the figures). To track your progress, output the move to the command line (“Move disk *x* from tower *y* to tower *z*”). You also need to call the *moveDisk* method so that the code animates. The API of *moveDisk* is explained in Example 12.10.
    3. Finally, you need to move all the disks from the *useTower* (temporary tower, “middle tower” on the figures) to the *toTower* (destination tower, “right tower” on the figures). Again, you call *recursiveTOfH*.

For example, if you run your program with three disks, and assuming the towers are labeled 0, 1, and 2 from left to right, the command line output of your method should read something like:

```
Move disk 1 from tower 0 to tower 2
Move disk 2 from tower 0 to tower 1
Move disk 1 from tower 2 to tower 1
Move disk 3 from tower 0 to tower 2
Move disk 1 from tower 1 to tower 0
Move disk 2 from tower 1 to tower 2
Move disk 1 from tower 0 to tower 2
```

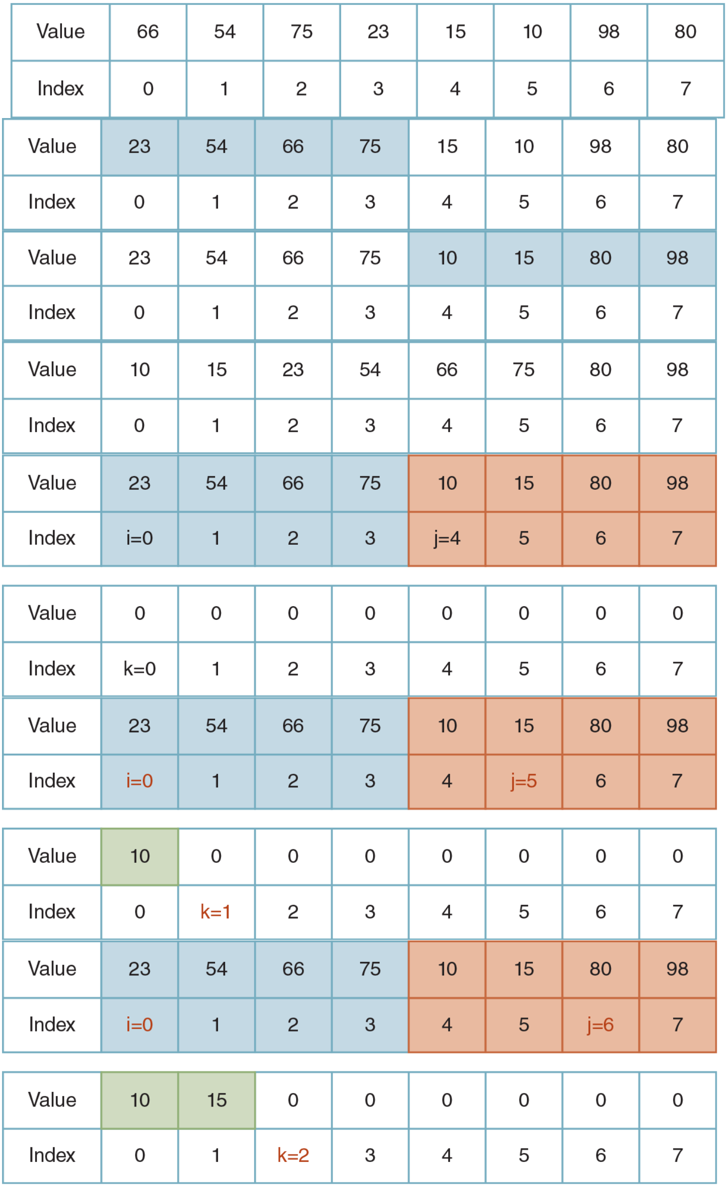
Troubleshooting

If your method implementation does not animate or animates incorrectly, check these items:

- Check the feedback on the output to see if your code violates the rules.
- Verify that you coded the first recursive call correctly.
- Verify that you coded the second recursive call correctly.

DISCUSSION QUESTIONS

1. What is the base case for the method?
2. As the number of disks increases, what happens to the time it takes for the method to run?



Value	66	54	75	23	15	10	98	80
Index	0	1	2	3	4	5	6	7

Value	23	54	66	75	15	10	98	80
Index	0	1	2	3	4	5	6	7

Value	23	54	66	75	10	15	80	98
Index	0	1	2	3	4	5	6	7

Value	10	15	23	54	66	75	80	98
Index	0	1	2	3	4	5	6	7

Value	23	54	66	75	10	15	80	98
Index	i=0	1	2	3	j=4	5	6	7

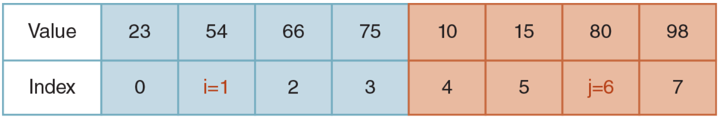
Value	0	0	0	0	0	0	0	0
Index	k=0	1	2	3	4	5	6	7

Value	23	54	66	75	10	15	80	98
Index	i=0	1	2	3	4	j=5	6	7

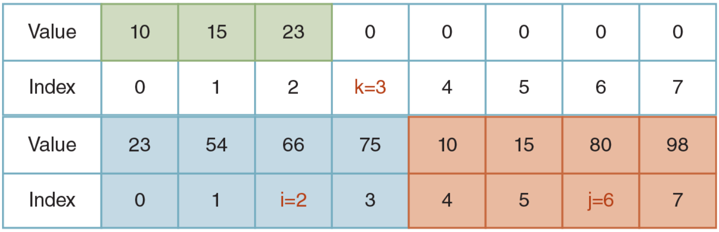
Value	10	0	0	0	0	0	0	0
Index	0	k=1	2	3	4	5	6	7

Value	23	54	66	75	10	15	80	98
Index	i=0	1	2	3	4	5	j=6	7

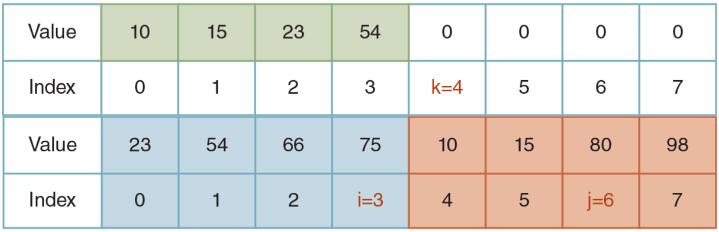
Value	10	15	0	0	0	0	0	0
Index	0	1	k=2	3	4	5	6	7



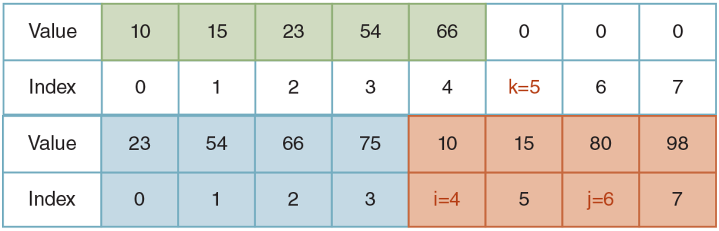
Value	23	54	66	75	10	15	80	98
Index	0	i=1	2	3	4	5	j=6	7



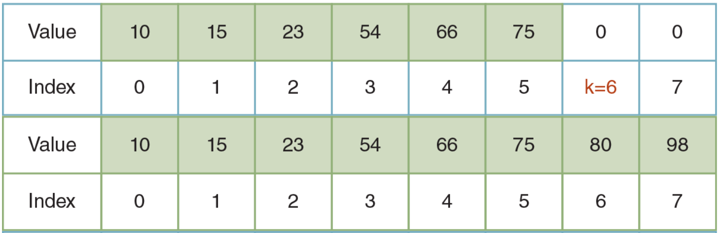
Value	10	15	23	0	0	0	0	0
Index	0	1	2	k=3	4	5	6	7
Value	23	54	66	75	10	15	80	98
Index	0	1	i=2	3	4	5	j=6	7



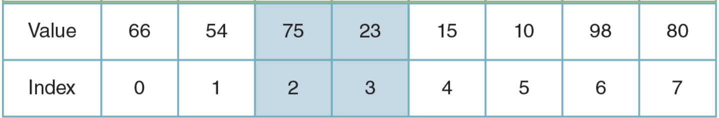
Value	10	15	23	54	0	0	0	0
Index	0	1	2	3	k=4	5	6	7
Value	23	54	66	75	10	15	80	98
Index	0	1	2	i=3	4	5	j=6	7



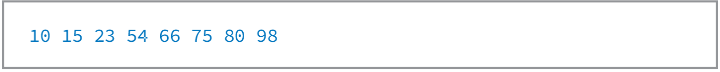
Value	10	15	23	54	66	0	0	0
Index	0	1	2	3	4	k=5	6	7
Value	23	54	66	75	10	15	80	98
Index	0	1	2	3	i=4	5	j=6	7



Value	10	15	23	54	66	75	0	0
Index	0	1	2	3	4	5	k=6	7
Value	10	15	23	54	66	75	80	98
Index	0	1	2	3	4	5	6	7



Value	66	54	75	23	15	10	98	80
Index	0	1	2	3	4	5	6	7



```
10 15 23 54 66 75 80 98
```

```
mergeSort( 0, 7 ) called
mergeSort( 0, 3 ) called
mergeSort( 0, 1 ) called
mergeSort( 0, 0 ) called
mergeSort( 1, 1 ) called
merge( 0, 0, 1, 1 ) called
mergeSort( 2, 3 ) called
mergeSort( 2, 2 ) called
mergeSort( 3, 3 ) called
merge( 2, 2, 3, 3 ) called
merge( 0, 1, 2, 3 ) called
mergeSort( 4, 7 ) called
mergeSort( 4, 5 ) called
mergeSort( 4, 4 ) called
mergeSort( 5, 5 ) called
merge( 4, 4, 5, 5 ) called
mergeSort( 6, 7 ) called
mergeSort( 6, 6 ) called
mergeSort( 7, 7 ) called
merge( 6, 6, 7, 7 ) called
merge( 4, 5, 6, 7 ) called
merge( 0, 3, 4, 7 ) called

10 15 23 54 66 75 80 98
```

12.8 Merge Sort

Selection sort, insertion sort, and bubble sort (discussed earlier in this text) all involve a double loop to sort an array. Two other sorting algorithms are **Merge Sort** and **Quick Sort**. They both use recursion. We study Merge Sort first. In the general case, it works as follows:

- Step 1: Merge sort the left half of the array (this is a recursive call)
- Step 2: Merge sort the right half of the array (this is a recursive call)
- Step 3: Merge the two sorted half arrays

For example, we will use merge sort to sort the following array:

Description

Step 1 is calling merge sort to sort the left half of the array;

after step 1, the array now looks like:

Description

Step 2 is calling merge sort to sort the right half of the array; after step 2, the array now looks like:

Description

At this point, the left and right halves of the array are sorted. In step 3, we merge them together so that the whole array is now sorted.

Description

To do so, we loop through all the elements of the left and right subarrays in parallel fashion, comparing them two at a time and placing them in the resulting sorted array. In doing so, we use three indices:

- i is the index of the value that we are processing in the left subarray
- j is the index of the value that we are processing in the right subarray
- k is the index, in the resulting sorted array, of the value that we are placing

That sequence is illustrated as follows. At the beginning, i is 0 (the first index of the left subarray, j is 4 (the first index of the right subarray), and k is 0 (the first index of the resulting array).

Description

We now process the elements at index i = 0 (23) and j = 4 (10). 10 is smaller, so we place 10 at index k (0) in the resulting array and increment j by 1 (to 5) and k by 1 (to 1).

Description

We now process the elements at index i = 0 (23) and j = 5

(15). 15 is smaller, so we place 15 at index k (1) in the resulting array and increment j by 1 (to 6) and k by 1 (to 2).

Description

We now process the elements at index i = 0 (23) and j = 6 (80). 23 is smaller, so we place 23 at index k (2) in the resulting array and increment i by 1 (to 1) and k by 1 (to 3).

Description

We now process the elements at index i = 1 (54) and j = 6 (80). 54 is smaller, so we place 54 at index k (3) in the resulting array and increment i by 1 (to 2) and k by 1 (to 4).

Description

We now process the elements at index i = 2 (66) and j = 6 (80). 66 is smaller, so we place 66 at index k (4) in the resulting array and increment i by 1 (to 3) and k by 1 (to 5).

Description

We now process the elements at index i = 3 (75) and j = 6 (80). 75 is smaller, so we place 75 at index k (5) in the resulting array and increment i by 1 (to 4) and k by 1 (to 6).

Description

At this point, i, which has value 4, has gone past the last index (3) of the left half of the array. We have finished processing all the values of the left half of the array. We loop through the remaining values of the right half of the array and place them in the same order (since they are already sorted) in the resulting array, which is now full and sorted.

Description

In this example, we finished processing the left half of the array first. In general, we do not know which half of the array we will finish processing first. That does not matter. Once we know that we finished processing one half of the array, we can simply loop through both halves of the array and place their elements in the resulting array in order. One of the two halves is empty; that loop will not execute.

At each recursive call, we call merge sort to sort a subarray within the original array. As the recursive calls are made, the sizes of the subarrays are getting smaller and smaller. Eventually, we end up making a recursive call on a subarray of only one element; that is the base case. There is nothing to do; a subarray of only one element is automatically sorted. Note that if the original array is empty, then there is nothing to do either.

A subarray of the original array is defined by two parameters: its starting index and its ending index. For example, a starting index of 2 and ending index of 3 define the subarray made up of the values 75 and 23, as shown below:

Description

Example 12.11 shows the implementation of merge sort (lines 16–37). In the general case (lines 28–34), we make a recursive call to sort the left half of the array (lines 29–30), then make another recursive call to sort the right half of the array (lines 31–32), and then call *merge* (lines 33–34) to merge the two halves. The *merge* method is shown at lines 39 to 94. We declare and instantiate a new array, *temp*, of the same size as the original array at line 54. At lines 59 to 73, we place the elements of the two halves of the array into the new array until one half of the array is fully processed. Note that we do not know which half that is. At lines 75 to 81, we flush the remaining elements of the left half of the array into *temp*, if necessary. At lines 83 to 89, we flush the remaining elements of the right half of the array into *temp*, if necessary. At lines 91 to 93, we copy the elements of *temp* back to the original array.

Example 12.11 Sorting an Array of Integers Using Merge Sort

```
/* Sorting an array using merge sort
  Anderson, Franceschi
*/

public class MergeSort
{
  public static void main( String [] args )
  {
    int [] numbers = { 66, 54, 75, 23, 15, 10, 98, 80 };
    mergeSort( numbers, 0, numbers.length - 1 );
    for ( int i = 0; i < numbers.length; i++ )
      System.out.print( numbers[i] + " " );
    System.out.println( );
  }

 /**
  * mergeSort method: sorts the subarray of arr
  * (from index start to index end, both included)
  * in ascending order
  * @param arr, an array of ints
  * @param start an int, the starting index of the subarray
  * @param end an int, the ending index of the subarray
  */
  public static void mergeSort( int [] arr, int start, int end )
  {
    if ( start < end ) // general case
    {
      int middle = ( start + end ) / 2;
      // sort the left part of the array
      mergeSort( arr, start, middle );
      // sort the right part of the array
      mergeSort( arr, middle + 1, end );
      // merge the 2 parts
      merge( arr, start, middle, middle + 1, end );
    }
    // else, base case, empty or 1 element array, already sorted
  }

 /**
  * merge method: merges, in ascending order,
  * the two subarrays of arr defined as follows:
  * left subarray: between indexes start1 and end1, both included
  * right subarray: between indexes start2 and end2, both included
  * @param arr, an array of ints
  * @param start1 an int, the starting index of the left subarray
  * @param end1 an int, the ending index of the left subarray
  * @param start2 an int, the starting index of the right subarray
  * @param end2 an int, the ending index of the right subarray
  */
  public static void merge( int [] arr, int start1, int end1,
                                        int start2, int end2 )
  {
    // merge the 2 sides into a temp array
    int [] temp = new int[arr.length]; // temporary array
    int i = start1;
    int j = start2;
    int k = start1;

    // generate merged subarray until one side is done
    while ( i <= end1 && j <= end2 )
    {
      if ( arr[i] < arr[j] )
      {
        temp[k] = arr[i];
        i++;
      }
      else
      {
        temp[k] = arr[j];
        j++;
      }
      k++;
    }

    // flush left subarray if necessary
    while ( i <= end1 )
    {
      temp[k] = arr[i];
      i++;
      k++;
    }

    // flush right subarray if necessary
    while ( j <= end2 )
    {
      temp[k] = arr[j];
      j++;
      k++;
    }

    // write back temp into arr
    for ( i = start1; i <= end2; i++ )
      arr[i] = temp[i];
  }
}
```

Description

At lines 9 to 13 of the *main* method, we call *mergeSort* to sort an array of integers. **Figure 12.21** shows the output of Example 12.11.

Figure 12.21 Output of Example 12.11

Description

If we want to trace all the recursive calls to *mergeSort* and the ensuing calls to *merge*, we can add the following statements after line 25 and after line 52, respectively:

```
System.out.println("mergeSort( " + start + ", " + end + " ) called");
System.out.println("merge( " + start1 + ", " + end1
                      + ", " + start2 + ", " + end2 + " ) called");
```

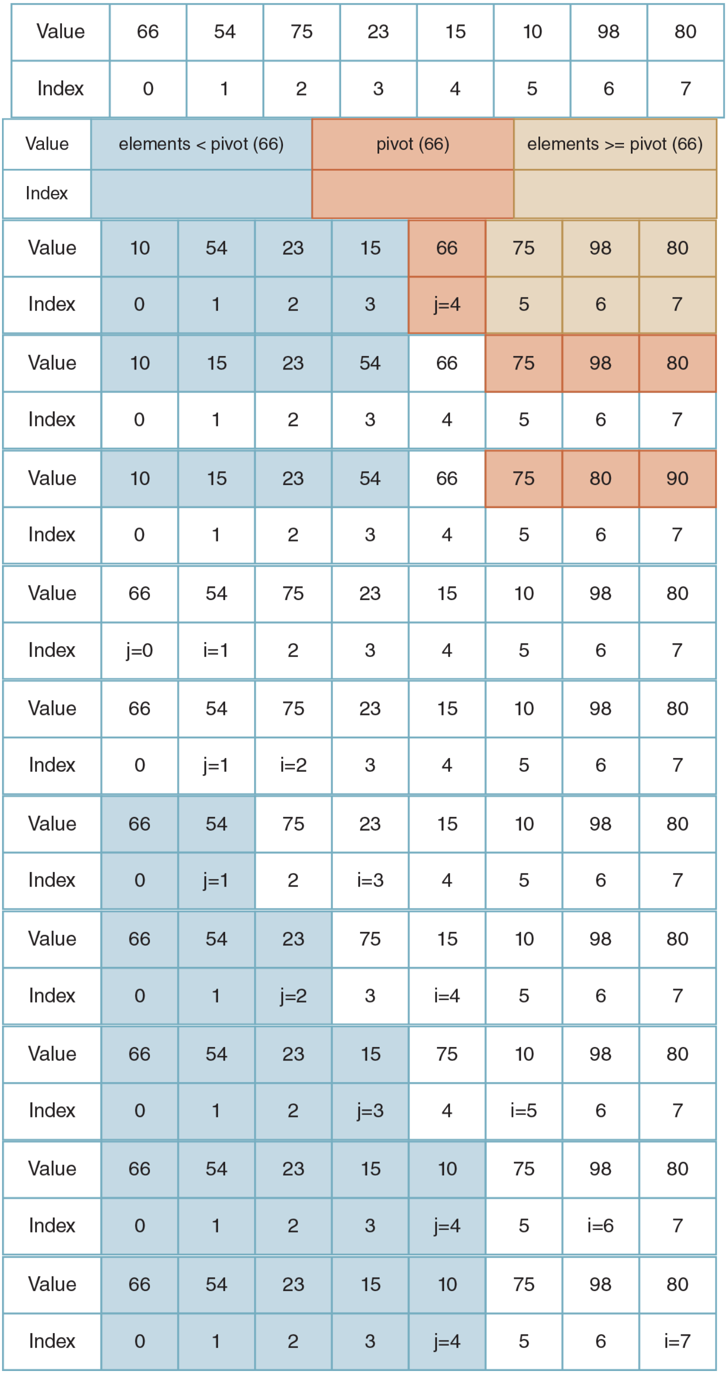
Description

Figure 12.22 shows the resulting output when we make the original call: *mergeSort(numbers, 0, 7)*. The calls resulting from step 1, step 2, and step 3 (as they apply to the whole array) are highlighted.

Figure 12.22 Output of Example 12.11 with Tracing Added

Description

We will later study the running time of merge sort and compare it to the running times of selection sort, insertion sort, and bubble sort.



Value	66	54	75	23	15	10	98	80
Index	0	1	2	3	4	5	6	7

Value	elements < pivot (66)	pivot (66)	elements >= pivot (66)
Index			

Value	10	54	23	15	66	75	98	80
Index	0	1	2	3	j=4	5	6	7

Value	10	15	23	54	66	75	98	80
Index	0	1	2	3	4	5	6	7

Value	10	15	23	54	66	75	80	90
Index	0	1	2	3	4	5	6	7

Value	66	54	75	23	15	10	98	80
Index	j=0	i=1	2	3	4	5	6	7

Value	66	54	75	23	15	10	98	80
Index	0	j=1	i=2	3	4	5	6	7

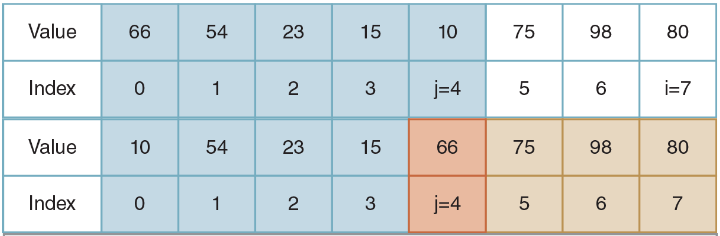
Value	66	54	75	23	15	10	98	80
Index	0	j=1	2	i=3	4	5	6	7

Value	66	54	23	75	15	10	98	80
Index	0	1	j=2	3	i=4	5	6	7

Value	66	54	23	15	75	10	98	80
Index	0	1	2	j=3	4	i=5	6	7

Value	66	54	23	15	10	75	98	80
Index	0	1	2	3	j=4	5	i=6	7

Value	66	54	23	15	10	75	98	80
Index	0	1	2	3	j=4	5	6	i=7



Value	66	54	23	15	10	75	98	80
Index	0	1	2	3	j=4	5	6	i=7
Value	10	54	23	15	66	75	98	80
Index	0	1	2	3	j=4	5	6	7

```
10 15 23 54 66 75 80 98
```

```
quickSort( 0, 7 ) called
partition( 0, 7 ) called
quickSort( 0, 3 ) called
partition( 0, 3 ) called
quickSort( 0, -1 ) called
quickSort( 1, 3 ) called
partition( 1, 3 ) called
quickSort( 1, 2 ) called
partition( 1, 2 ) called
quickSort( 1, 0 ) called
quickSort( 2, 2 ) called
quickSort( 4, 3 ) called
quickSort( 5, 7 ) called
partition( 5, 7 ) called
quickSort( 5, 4 ) called
quickSort( 6, 7 ) called
partition( 6, 7 ) called
quickSort( 6, 6 ) called
quickSort( 8, 7 ) called
10 15 23 54 66 75 80 98
```

12.9 Quick Sort

Quick sort, like merge sort, also uses recursion. In the general case, it works as follows:

- Step 1: Use one array element, called the pivot, to partition the array into two parts, left and right. All the elements on the left side are smaller than the pivot, and all the elements on the right side are greater than or equal to the pivot.
- Step 2: Quick sort the left part of the array (this is a recursive call)
- Step 3: Quick sort the right part of the array (this is a recursive call)

We use quick sort to sort the following array:

Description

In step 1, we partition the array around one of the array elements called the pivot. Because we have no information about the array, the easiest way is to choose the first element of the array, here 66, as the pivot. The *partition* method, coded at lines 37 to 59 of **Example 12.12**, moves the pivot element somewhere in the array and reorganizes the array as follows:

Example 12.12 Sorting an Array of Integers Using Quick Sort

```java
/* Sorting an array using quick sort
   Anderson, Franceschi
*/

public class QuickSort
{
  public static void main( String [] args )
  {
    int [] numbers = { 66, 54, 75, 23, 15, 10, 98, 80 };
    quickSort( numbers, 0, numbers.length - 1 );
    for ( int i = 0; i < numbers.length; i++ )
      System.out.print( numbers[i] + " " );
    System.out.println( );
  }

  /**
  * quickSort method: sorts the subarray of arr
  * (from index start to index end, both included)
  * in ascending order
  * @param arr, an array of ints
  * @param start an int, the starting index of the subarray
  * @param end an int, the ending index of the subarray
  */
  public static void quickSort( int [] arr, int start, int end )
  {
    if ( end > start ) // general case, >= 2 elements
    {
      int pivotIndex = partition( arr, start, end );
      // quick sort left subarray
      quickSort( arr, start, pivotIndex - 1 );
      // quick sort right subarray
      quickSort( arr, pivotIndex + 1, end );
    }
    // else, 0 or 1 element ==> subarray is sorted, do nothing
  }

  /**
  * partition method: partitions the subarray of arr
  * (from index start to index end, both included)
  * around the pivot, arr[start]
  * @param arr, an array of ints
  * @param start an int, the starting index of the subarray
  * @param end an int, the ending index of the subarray
  */
  public static int partition( int [] arr, int start, int end )
  {
    int j = start;
    int pivot = arr[start];
    for ( int i = start + 1; i <= end; i++ )
    {
      if ( arr[i] < pivot )
      {
        j++;
        swap( arr, i, j );
      }
    }
    swap( arr, start, j );
    return j;
  }

  /**
  * swap method: swaps the elements of arr
  * at index i and index j
  * @param arr, an array of ints
  * @param i an int, a valid index of arr
  * @param j an int, a valid index of arr
  */
  public static void swap( int [] arr, int i, int j )
  {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }
}
```

Description

Description

For this example, this is what the array will look like after the partition method executes:

Description

The array elements left of the pivot (10, 54, 23, and 15) are all smaller than the pivot.

The array elements right of the pivot (75, 98, 80) are all greater than or equal to the pivot.

The pivot is in its final position. The elements left of the pivot and right of the pivot are not sorted. Note that the pivot can end up being anywhere, including in the first or last position. Ideally, we would want the pivot to be in the middle of the array so that the array is partitioned evenly. However, there is no guarantee of that happening.

After calling quick sort to sort the left side of the array (step 2), the array now looks like:

Description

After calling quick sort to sort the right side of the array (step 3), the array is now sorted and looks like:

Description

Below is a sequence showing the execution of *partition* when originally called.

The pivot is 66, the first element of the array, start is 0, and end is 7. The indexes i and j are equal to 1 and 0, respectively.

Description

The element at index i, 54, is less than 66, so we increment j by 1 and swap the elements at indexes i (1) and j (1), which, in this case, is a no-op.

Description

In the second iteration of the loop (i = 2), we compare 75, the element at index 2, with 66, the pivot. 75 is not smaller than 66, so we do nothing.

Description

In the third iteration of the loop (i = 3), we compare 23, the element at index 3, with 66, the pivot. 23 is smaller than 66, so we increment j by 1 and swap the elements at indexes i and j, that is, 3 and 2.

Description

Note that the index j tracks the index of the last element of the left side of the array, which contains all the elements less than the pivot.

In the fourth iteration of the loop (i = 4), we compare 15, the element at index 4, with 66, the pivot. 15 is smaller than 66, so we increment j by 1 and swap the elements at indexes i and j, that is, 4 and 3.

Description

In the fifth iteration of the loop (i = 5), we compare 10, the element at index 5, with 66, the pivot. 10 is smaller than 66, so we increment j by 1 and swap the elements at indexes i and j, that is, 5 and 4.

Description

In the sixth iteration of the loop (i = 6), we compare 98, the element at index 6, with 66, the pivot. 98 is not smaller than 66, so we do nothing.

Description

In the seventh and last iteration of the loop (i = 7), we compare 80, the element at index 7, with 66, the pivot. 80 is not smaller than 66, so we do nothing.

Description

After the loop, we swap the elements at index 0 (the value of start) and 4 (the value of j) so that the pivot, 66, is placed exactly between the left side of the array (all those elements are less than the pivot) and the right side of the array (all those elements are greater than or equal to the pivot).

Description

Example 12.12 shows the implementation of quick sort (lines 16 to 35). The *quickSort* method sorts a subarray of its array parameter, *arr*. Like with merge sort, a subarray of the original array is defined by two parameters: its starting index and its ending index. Thus, the *quickSort* method sorts the subarray of *arr* whose elements are between the *start* and *end* indices, both included. At line 10, we sort the array *numbers* by specifying a *start* value of 0 and an *end* value of *numbers.length – 1*, identifying the whole array *numbers* as the subarray.

In the general case (lines 28–32), we first partition the array at line 28. Note that the *partition* method returns an integer, the index of the pivot after *partition* has finished executing. In this example, when *partition* is first called on the whole array, the index returned by *partition* is 4. That value is used in the two recursive calls to *quickSort* at lines 30 and 32. We make a recursive call to sort the left part of the array (lines 29–30), then make another recursive call to sort the right part of the array (lines 31–32).

The *partition* method is coded at lines 37 to 59. The pivot is defined, at line 48, as the first element of the subarray being sorted. At lines 49 to 56, we use a loop to partition the subarray using the pivot, but without moving the pivot. At line 57, we swap the pivot with the last element of the left part of the array. At this point, the pivot has moved to its final position. All the elements on the left side of the pivot are smaller than the pivot, and all the elements on the right side of the pivot are greater than or equal to the pivot.

At line 10 of the *main* method, we call *quickSort* to sort an array of integers named *numbers*, and output the sorted array at lines 11 to 12. **Figure 12.23** shows the output of Example

12.12.

Figure 12.23 Output of Example 12.12

Description

If we want to trace all the recursive calls to *quickSort* and the ensuing calls to *partition*, we can add the following statements after line 25 and after line 46, respectively:

```
System.out.println("quickSort( " + start + ", " + end + " ) called");
System.out.println("partition( " + start + ", " + end + " ) called");
```

Description

Figure 12.24 shows the resulting output when we make the original call: *quickSort(numbers, 0, 7)*. The calls resulting from step 1, step 2, and step 3 are highlighted.

Figure 12.24 Output of Example 12.12 with Tracing Added

Description

We will study the running time of quick sort and compare it to the other sorting algorithms we have studied later.

12.10 Recursion Versus Iteration

Recursion and iteration are different approaches to solving a problem.

- A recursive function is implemented using decision constructs (e.g., *if/else* statements) and repeatedly calls itself.
- An iterative function is implemented with looping constructs (e.g., *while* or *for* statements) and repeatedly executes the loop.

Most programmers would not use recursion to print “Hello World” *n* times; they would simply use the following *for* loop:

```
for ( int i = 0; i < n; i++ )
      System.out.println( "Hello World" );
```

Similarly, a factorial method can easily be coded using iteration.

However, other problems, such as the Towers of Hanoi and the binary search, are more easily coded using recursion rather than iteration.

SOFTWARE ENGINEERING TIP

Readability and maintenance are important considerations when coding a method. If running times are equivalent and a recursive implementation is easier to understand than the equivalent iterative implementation, choose recursion over iteration; otherwise, iteration is generally preferred.

Often, the recursive solution to a problem is more elegant and easier to understand than an equivalent iterative solution. The main difficulty in coding a recursive method is the problem-solving part. The implementation, using Java or any other programming language, is easier than the equivalent code that uses iteration.

Another consideration when deciding to use recursion or iteration is the efficiency of the method at execution time. This is called the running time of the method, and is often measured in order of magnitude as a function of the size of the input.

For instance, for the Hello World example, the input is *n*. Using iteration, we will execute the loop *n* times, and the test condition of the *for* statement will be executed ($n + 1$) times. When the running time of a method can be expressed as *n* multiplied by a constant value ($n \times c$), we say that the order of magnitude of its running time is *n*; we say it is "big-Oh" of *n* or $O(n)$.

Using recursion, and considering Example 12.1, the *printHelloWorldNTimes* method will call itself *n* times before reaching the base case ($n = 0$), for which the method does nothing. At each recursive call, the method performs a small and finite number of operations: it tests for *n* being greater than 0, and then prints "Hello World" once before calling itself; so it performs two operations before calling itself. Since it calls itself *n* times overall, the approximate running time of the method is 2 * *n*, and therefore the order of magnitude of its running time is also $O(n)$ (orders of magnitude ignore constant factors).

So in this case, the running times of iteration and recursion are of the same order of magnitude. However, the overhead associated with all the recursive method calls will add to the running time. That will make recursion slower than iteration, which is often the case.

Description

© Lois GoBe/Shutterstock.

CHAPTER REVIEW

12.11 Chapter Summary

- The idea of recursion is to convert or reduce a bigger problem to a smaller, similar problem. The relationship between the bigger problem and the smaller problem is called the general case.
- By reducing the size of a problem to a smaller problem recursively, we eventually arrive at a small problem that is easy to solve. That small problem is called the base case.
- Solving a problem using recursion typically involves coding a recursive method.
- A recursive method
    - can be a *static* method or an instance method,
    - can take 0, 1, or more parameters, and
    - can be a *void* or a value-returning method.
- A recursive method calls itself.
- Problem solving using recursion involves two steps: generating a recursive formulation of the problem for the general case, and solving the base case(s).
- There can be one or more base cases.

- Most base cases are simple, but some can be more complex.
- Most general cases are simple, but some can be more complex.
- A recursive method calls itself repeatedly until a base case is reached.
- A recursive method typically includes an *if/else* statement that tests for the base case.
- If the recursive method does not test for the base case, calling the method will typically result in a stack overflow run-time error.
- Recursion is typically an alternative to iteration. The coding of a recursive method is typically compact and elegant. However, a recursive method may not be as efficient as its iterative equivalent.

12.12 Exercises, Problems, and Projects

12.12.1 Multiple Choice Exercises

1. A recursive method

   - is always a *static* method.
   - is never a *static* method.
   - may or may not be *static*.

5. A recursive method

   - is always a method with a *void* return value.
   - is always a value-returning method.
   - can be either of the above.

9. When formulating a recursive solution, what should we consider?

   - base cases and general case
   - base cases only
   - general case only

13. A recursive method

    - is a method containing a loop.
    - calls itself.
    - is part of the *java.recursion* package.

17. When coding a class that includes a recursive method, we need to import the *java.recursion* package.

    - true
    - false

20. If the base case of a recursive method is not taken into account when coding the method, the likely outcome is

    - a compiler error.
    - a run-time error.
    - no error.

24. If there are several base cases in a recursive method, omitting the code for one of them will result in

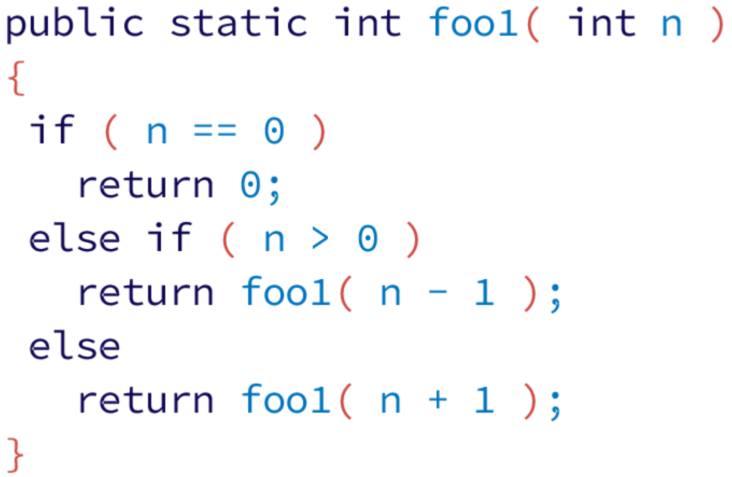
    - a compiler error.
    - a run-time error.
    - no error.

28. If a recursive method makes a recursive call with the same argument that it was passed, the likely outcome is

    - a compiler error.
    - a run-time error.
    - no error.

12.12.2 Reading and Understanding Code

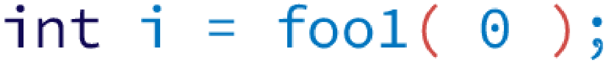
For Questions 9 to 11, consider the following method:



```
public static int foo1( int n )
{
  if ( n == 0 )
    return 0;
  else if ( n > 0 )
    return foo1( n - 1 );
  else
    return foo1( n + 1 );
}
```

Description

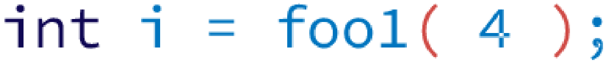
1. What is the value of *i* after the following code is executed?



```
int i = foo1( 0 );
```

Description

2. What is the value of *i* after the following code is executed?

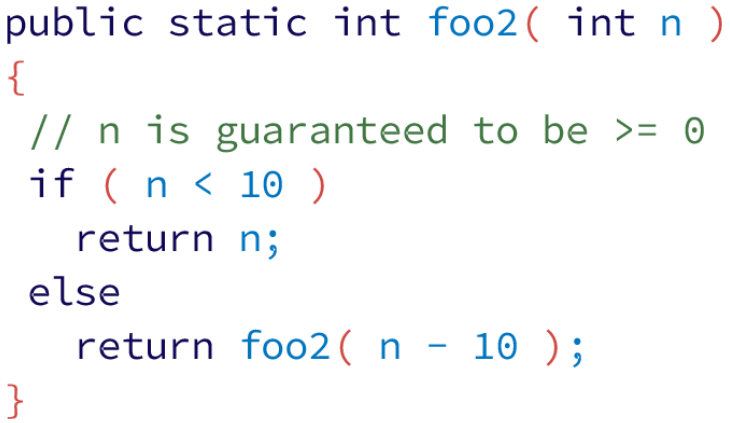


```
int i = foo1( 4 );
```

Description

3. What does the *foo1* method do?

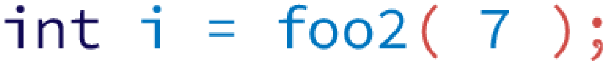
For Questions 12 to 15, consider the following method:



```
public static int foo2( int n )
{
 // n is guaranteed to be >= 0
 if ( n < 10 )
   return n;
 else
   return foo2( n - 10 );
}
```

Description

1. What is the value of *i* after the following code is executed?



```
int i = foo2( 7 );
```

Description

2. What is the value of *i* after the following code is executed?

```
int i = foo2( 13 );
```

Description

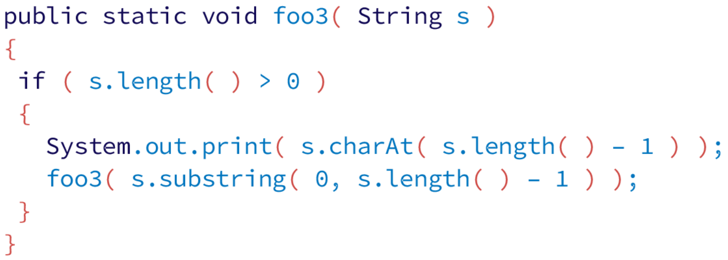
3. What is the value of *i* after the following code is executed?

```
int i = foo2( 65 );
```

Description

4. What does the *foo2* method return when the argument is a positive integer?

For Questions 16 to 19, consider the following method:



```
public static void foo3( String s )
{
 if ( s.length( ) > 0 )
 {
   System.out.print( s.charAt( s.length( ) - 1 ) );
   foo3( s.substring( 0, s.length( ) - 1 ) );
 }
}
```

Description

1. What is the output of the following code?

```
foo3( "" );
```

2. What is the output of the following code?

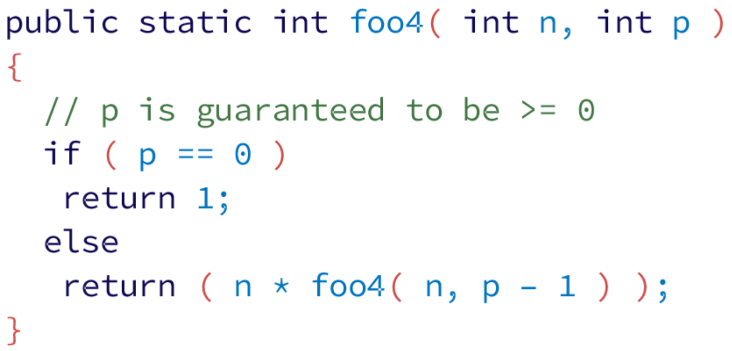
```
foo3( "Hi" );
```

3. What is the output of the following code?

```
foo3( "Hello" );
```

4. What does the *foo3* method do?

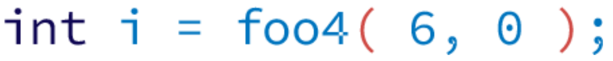
For Questions 20 to 23, consider the following method:



```
public static int foo4( int n, int p )
{
  // p is guaranteed to be >= 0
  if ( p == 0 )
   return 1;
  else
   return ( n * foo4( n, p - 1 ) );
}
```

Description

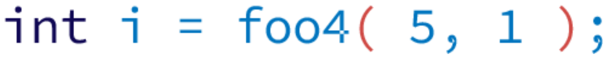
1. What is the value of *i* after the following code is executed?



```
int i = foo4( 6, 0 );
```

Description

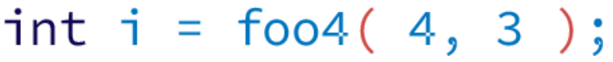
2. What is the value of *i* after the following code is executed?



```
int i = foo4( 5, 1 );
```

Description

3. What is the value of *i* after the following code is executed?



```
int i = foo4( 4, 3 );
```

Description

4. What does the *foo4* method return as a function of its two parameters, *n* and *p*?

12.12.3 Fill In the Code

1. This recursive method returns the number of times a given character is found in a *String*.

```
public static int foo( String s, char c )
{
  if ( s.length( ) == 0 )
    return 0;
  else
  {
    // your code goes here

  }
}
```

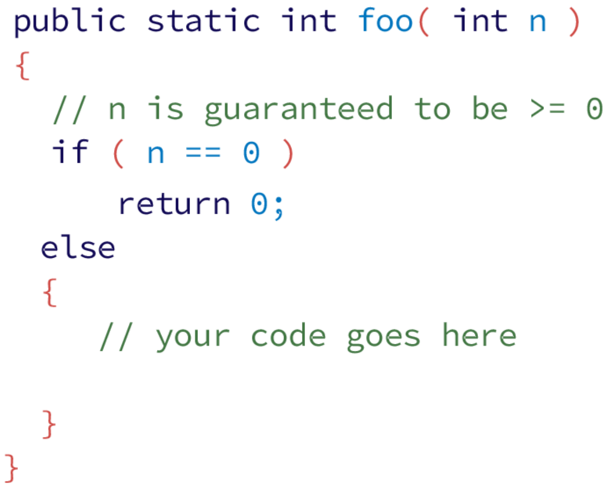
   Description

2. This recursive method returns “even” if the length of a given *String* is even, and “odd” if the length of the *String* is odd.

```
public static String foo( String s )
{
  if ( s.length( ) == 0 )
      return "even";
  else if ( s.length( ) == 1 )
      return "odd";
  else
       // your code goes here
}
```

   Description

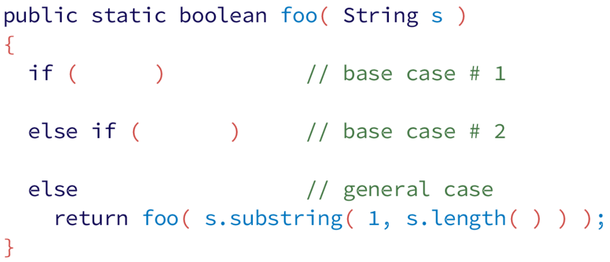
3. This recursive method returns the sum of all the integers from 0 to a given number.



```
public static int foo( int n )
{
  // n is guaranteed to be >= 0
  if ( n == 0 )
     return 0;
 else
 {
    // your code goes here

 }
}
```

Description

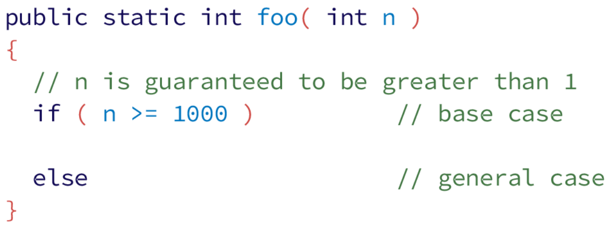
4. This recursive method returns *true* if its *String* parameter contains the characters *A* and *B* in consecutive locations; otherwise, it returns *false*.



```
public static boolean foo( String s )
{
  if (      )           // base case # 1

  else if (       )     // base case # 2

  else                  // general case
    return foo( s.substring( 1, s.length( ) ) );
}
```

Description

5. This recursive method squares a number until the result is greater than or equal to 1000, then returns the result. For instance, *foo(10)* returns 10000, *foo(6)* returns 1296, and *foo(1233)* returns 1233.

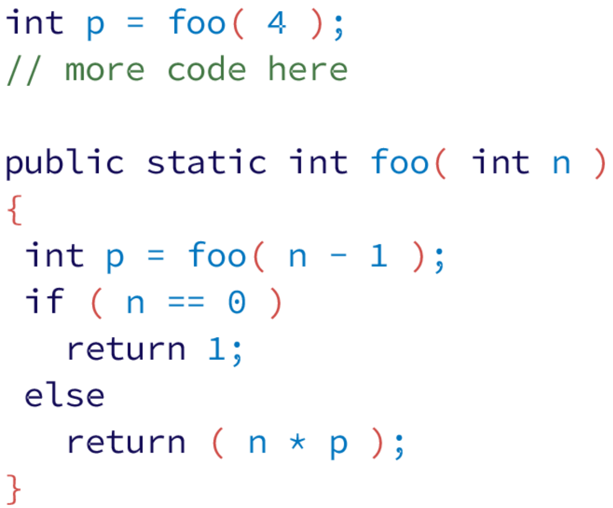


```
public static int foo( int n )
{
  // n is guaranteed to be greater than 1
  if ( n >= 1000 )          // base case

  else                      // general case
}
```

Description

12.12.4 Identifying Errors in Code

1. You coded the following in the file *Test.java*. Where is the error?



```
int p = foo( 4 );
// more code here

public static int foo( int n )
{
 int p = foo( n - 1 );
 if ( n == 0 )
   return 1;
 else
   return ( n * p );
}
```

   Description

2. You coded the following method. Where is the error?

```
public static double foo( int n )
{
 if ( n == 0 )
   return 1.0;
 else if ( n < 0 )
   return foo( n - 1 );
 else
   return foo( n + 1 );
}
```

Description

3. You coded the following method. Where is the error?

```
public static boolean foo( int n )
{
 // n is guaranteed to be >= 0
 if ( n == 0 )
   return true;
 else
   foo( n - 1 );
}
```

Description

4. You coded the following method. Where is the error?

```
public static boolean foo( int n )
{
 // n is guaranteed to be >= 0
 if ( n == 0 )
   return true;
 else
   return foo( n );
}
```

Description

12.12.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. You coded the following in the file *Test.java*:

```
// inside main
System.out.println( foo( 5 ) );
// more code here

public static int foo( int n )
{
 return ( n * foo( n - 1 ) );  // line 15
}
```

   Description
   The code compiles, but when it runs, you get the following output:

```
Exception in thread "main" java.lang.StackOverflowError
at Test.foo(Test.java:15)
at Test.foo(Test.java:15)
at Test.foo(Test.java:15)
....
```

   Description
   Explain what the problem is and how to fix it.

2. You coded the following in the file *Test.java*:

```
// inside main
System.out.println( foo( 5 ) );
// more code here

public static int foo( int n )
{
 if ( n == 0 )
   return foo( 0 );  // line 15
 else
   return ( n * foo( n - 1 ) );
}
```

Description

The code compiles, but when it runs, you get the following output:

```
Exception in thread "main" java.lang.StackOverflowError
at Test.foo(Test.java:15)
at Test.foo(Test.java:15)
at Test.foo(Test.java:15)
....
```

Description

Explain what the problem is and how to fix it.

3. You coded the following in the file *Test.java*:

```
// inside main
System.out.println( foo( 5 ) );
// more code here

public static int foo( int n ) // line 9
{
 if ( n == 0 )
   return 1;
 else
   System.out.println( n * foo( n - 1 ) );
}         // line 15
```

Description

At compile time, you get the following error:

```
Test.java:15: error: missing return statement
}        // line 15
^
1 error
```

Description

Explain what the problem is and how to fix it.

4. You coded the following in the file *Test.java*:

```
// inside main
System.out.println( foo( 5 ) );
// more code here

public static int foo( int n )
{
 if ( n == 0 )
   return 1;
 else
   return ( foo( n ) * ( n - 1 ) );
}
```

Description

The code compiles, but when it runs, you get the following message, repeated many times, before finally stopping:

```
Exception in thread "main" java.lang.StackOverflowError
     at Test.foo(Test.java:15)
     at Test.foo(Test.java:15)
     at Test.foo(Test.java:15)
```

Description

Explain what the problem is and how to fix it.

5. You coded the following in the file *Test.java*:

```
// inside main
System.out.println( foo( "Hello" ) ); // line 6
// more code here
public static int foo( String s )     // line 9
{
 if ( s.length( ) == 0 )
   return 0;
 else
   return ( 1 +
        foo( s.substring( 0, s.length( ) - 2 ) ) ); // line 15
   }
```

Description

The code compiles, but when it runs, you get the following output:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: begin 0,
end -1, length 1
   at java.base/java.lang.String.checkBoundsBeginEnd(String.java:4601)
   at java.base/java.lang.String.substring(String.java:2704)
   at Test.foo(Test.java:15)
   at Test.foo(Test.java:15)
   at Test.foo(Test.java:15)
   at Test.main(Test.java:6)
```

Description

Explain what the problem is and how it happens.

6. You coded the following in the file *Test.java*:

```
// inside main
System.out.println( foo( "Hello" ) ); // line 6
// more code here

public static int foo( String s ) // line 9
{
 if ( s.length( ) == 0 ) // line 11
   return 0;
 else
 {
    String temp = null;
    if ( s.length( ) > 1 )
         temp = s.substring( 0, s.length( ) - 1 );
    return ( 1 + foo( temp ) ); // line 18
 }
}
```

Description

The code compiles, but when it runs, you get the following output:

```
Exception in thread "main" java.lang.NullPointerException:
Cannot invoke "String.length( )" because <parameter1> is
null
        at Test.foo(Test.java:11)
        at Test.foo(Test.java:18)
        at Test.foo(Test.java:18)
        at Test.foo(Test.java:18)
        at Test.foo(Test.java:18)
        at Test.foo(Test.java:18)
        at Test.main(Test.java:6)
```

Description

Explain what the problem is and how to fix it.

12.12.6 Write a Short Program

1. Using recursion, write a program that takes a word as an input and outputs that word backward.
2. Using recursion, write a program that keeps prompting the user for a word containing a $ character. As soon as the user inputs a word containing the $ character, you should output that word and your program will terminate.
3. Using recursion, write a program that takes a word as an input and outputs that word with all characters separated by a space.
4. Using recursion, write a program that takes a word as an input and outputs the number of times the letter *a* is found in that word.
5. Using recursion, write a program that takes an integer value as an input and outputs the Fibonacci value for that number. The Fibonacci value of a number is defined as follows:

   ```
   Fib( 1 ) = 1
   Fib( 2 ) = 1
   Fib( n ) = Fib ( n - 1 ) + Fib( n - 2 ) for n >= 3
   ```

   Description
6. Using recursion, write a program that takes a positive number as an input and keeps dividing that number by 3 until the result is less than 1, at which time output that result.
7. Using recursion, write a program that takes 10 numbers as inputs and outputs the minimum of these numbers.
8. Using recursion, write a program that takes 10 words representing Internet addresses as inputs and outputs the number of words containing .edu.
9. Rewrite Example 12.9, *RecursiveBinarySearch*, using an array sorted in descending order.

12.12.7 Programming Projects

1. Write a class with just one instance variable, a *String* representing a binary number. Write a recursive method taking only one parameter that converts that binary number to its decimal equivalent. Your program should include a client class to test your class.
2. Write a class with just one instance variable, an *int*. Your constructor should take an *int* as its only parameter. Write a recursive method that checks if that *int* is a multiple of 5. Your program should include a client class to test your class.
3. Write a class with just one instance variable, a *String* representing some HTML code. Your constructor should take a file name as its only parameter (you will need to make up some sample HTML files to test your program). Write a recursive method returning the number of occurrences of a specified character in the HTML *String*. Your program should include a client class to test your class. In particular, call the recursive method to check whether the sample files contain an equal number of < and > characters.
4. Write a class with just one instance variable, a *String* representing a password. Write a recursive method to check if the password contains at least one character that is a digit (0 to 9). Your program should include a client class to test your class.
5. Write a class with two instance variables, representing the same password. Write a recursive method that checks if both passwords are equal. Your program should include a client class to test your class.

6. Write a class with two instance variables, representing an old password and a new password. Write a recursive method that returns the number of places where the two passwords have different characters. The passwords can have different lengths. Write another, nonrecursive method returning whether the two passwords are sufficiently different. The method takes an *int* parameter indicating the minimum number of differences that qualify the passwords as being sufficiently different. Your program should include a client class to test your class.
7. Write a class with just one instance variable, an integer array. Your constructor should take an integer array as its only parameter. Write a recursive method that returns the sum of all elements in the array. Your program should include a client class to test your class.
8. Write a class with just one instance variable, an integer array. Your constructor should take an integer array as its only parameter. Write a recursive method that returns the maximum value of all the elements in the array. Your program should include a client class to test your class.
9. Write a class with the functionality of checking a list of names to determine whether the same name is present in two consecutive locations. The method solving that problem should be recursive. Your program should include a client class to test your class.
10. A professor has a policy to give at least one A in his or her class. Write a class that encapsulates that idea, including a recursive method checking for at least one A in a set of grades. Your program should include a client class to test your class.
11. Write a class with just one instance variable, an array

representing grades between 0 and 100. Your constructor should take an array as its only parameter. Write a recursive method that returns the average of all grades. Your program should include a client class to test your class.

12. Write a class with just one instance variable, an *int*. Your constructor should take an *int* as its only parameter. Write a recursive method that converts that *int* to a *String* representing that number in binary. Your program should include a client class to test your class.
13. Write a class potentially representing a *String* of binary digits (0s and 1s). Your constructor should take a *String* as its only parameter (that *String* may contain only 0s and 1s, or it may not). Write a recursive method that checks whether that *String* contains 0s and 1s only. Write another recursive method that converts that *String* to its decimal equivalent. Your method should be different from the one in Exercise 48: it should take two parameters, the *String* representing the binary number, and an *int* representing an exponent. Your program should include a client class to test your class.
14. Add a *static* method to the *Rational* class that is discussed and implemented earlier in the book. That *static* method sorts an array of *Rational* objects in ascending order using merge sort. Add some code in the main to test your method.
15. Add a *static* method to the *Auto* class that is discussed and implemented earlier in the book. That *static* method sorts an array of *Auto* objects in ascending order using quick sort. *Auto* objects are sorted in alphabetical order using their *model* instance variable. Add some code in the main

to test your method.

16. Write a class that includes a recursive method that converts a prefix arithmetic expression to an infix arithmetic expression. You should assume that a prefix arithmetic expression uses this syntax:
operator integerl integer2 integer3 …
The corresponding infix arithmetic expression is:

integer1 operator integer2 operator integer3 …

The operator can be +, –, *, or /.
For example, with this prefix arithmetic expression:

$$-6\ 123\ 45$$

your method should return the following infix arithmetic expression:

$$6 - 123 - 45$$

And for this prefix arithmetic expression:

$$+76543\ 12\ 4\ 5\ 6\ 7$$

Your method should return the following infix arithmetic expression:

$$76543 + 12 + 4 + 5 + 6 + 7$$

For simplicity, assume that there is just one space between the operator and the first integer and between the integers. Look up the documentation for the *Matcher* and *Pattern* classes and regular expressions. Your program should include a client class to test your program.

12.12.8 Technical Writing

1. Think of an example of a problem, different from the chapter problems, which can be solved by an iterative formulation and a recursive formulation. Discuss which one you would prefer to code and why.

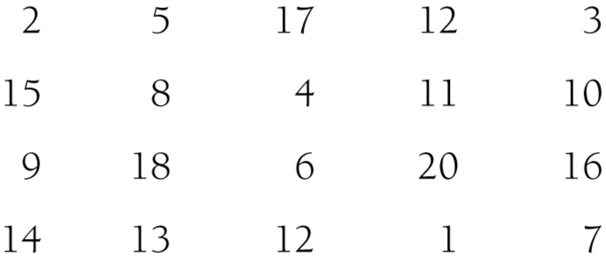
12.12.9 Group Projects (for a group of 1, 2, or 3 students)

1. Consider a rectangular grid of integers. We are interested in computing recursively the largest sum of any path from a top position to a bottom position. A valid path is defined as follows:

    - It should start at a number in the top row and end at a number in the bottom row.
    - It should include a number in every row.
    - From row *i* to row (*i* + 1), a valid path can be created:

- down vertically (in the same column)
- down diagonally one column to the left (if possible)
- down diagonally one column to the right (if possible)

For instance, let’s assume we have the following rectangle of numbers:



2	5	17	12	3
15	8	4	11	10
9	18	6	20	16
14	13	12	1	7

Note: Your program should accept any positive number at any spot within the rectangle.
Examples of valid paths are:
2 → 8 → 18 → 14
17 → 4 → 18 → 14
5 → 4 → 20 → 12
In this example, the path generating the largest sum is:
17 → 11 → 20 → 12 for a total of 17 + 11 + 20 + 12 = 60
Your program should accept from the user a rectangle of integers; to keep it simple, you can limit the size of the rectangle to a maximum of 10 columns by 20 rows. Your program should, recursively, compute and output the path that generates the largest sum.

© Lois GoBe/Shutterstock.

CHAPTER 13

An Introduction to Data Structures

CHAPTER CONTENTS

Introduction
13.1 Linked Lists
    13.1.1 Linked-List Concepts and Structure
    13.1.2 Linked-List Basics
    13.1.3 Methods of a Linked List
    13.1.4 Testing a Linked-List Class
13.2 Linked Lists of Objects
    13.2.1 A Linked-List Shell
    13.2.2 Generating an Exception
    13.2.3 Other Methods of a Linked List
    13.2.4 Testing a Linked-List Class
13.3 Implementing a Stack Using a Linked List
13.4 Implementing a Queue Using a Linked List
13.5 Array Representation of Stacks
13.6 **Programming Activity 1: Writing Methods for a**

Stack Class
13.7 Array Representation of Queues
13.8 Sorted Linked Lists
13.9 **Programming Activity 2: Writing *Insert* and *Delete* Methods for a Sorted Linked List**
13.10 Doubly Linked Lists
13.11 Linked Lists Using Generic Types
13.12 Recursively Defined Linked Lists
13.13 Chapter Summary
13.14 Exercises, Problems, and Projects
    13.14.1 Multiple Choice Exercises
    13.14.2 Reading and Understanding Code
    13.14.3 Fill In the Code
    13.14.4 Identifying Errors in Code
    13.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
    13.14.6 Write a Short Program
    13.14.7 Programming Projects
    13.14.8 Technical Writing
    13.14.9 Group Project

Introduction

As our programs execute, we often need a means to organize data in memory. Arrays are a convenient method to store multiple variables of the same data type. *ArrayLists* improve on arrays by dynamically expanding, as needed.

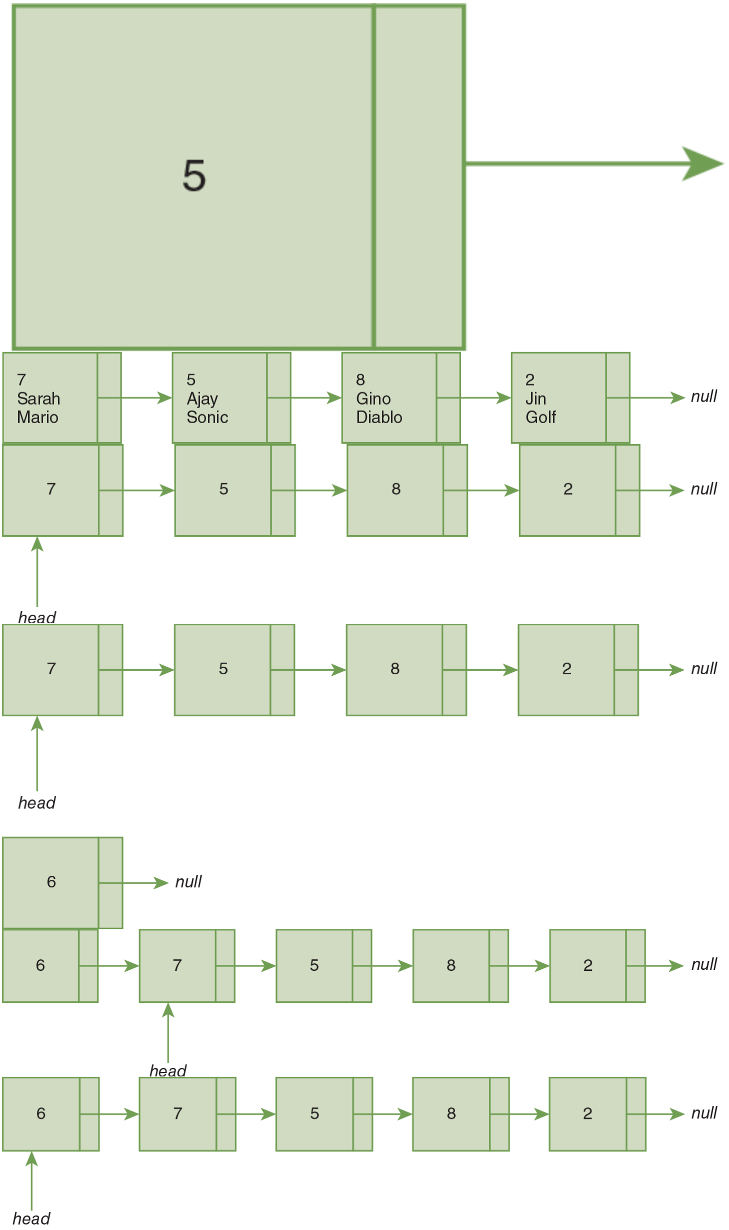
In fact, arrays and *ArrayLists* are just two examples of **data structures**, which are methodologies a program uses to store its data in memory.

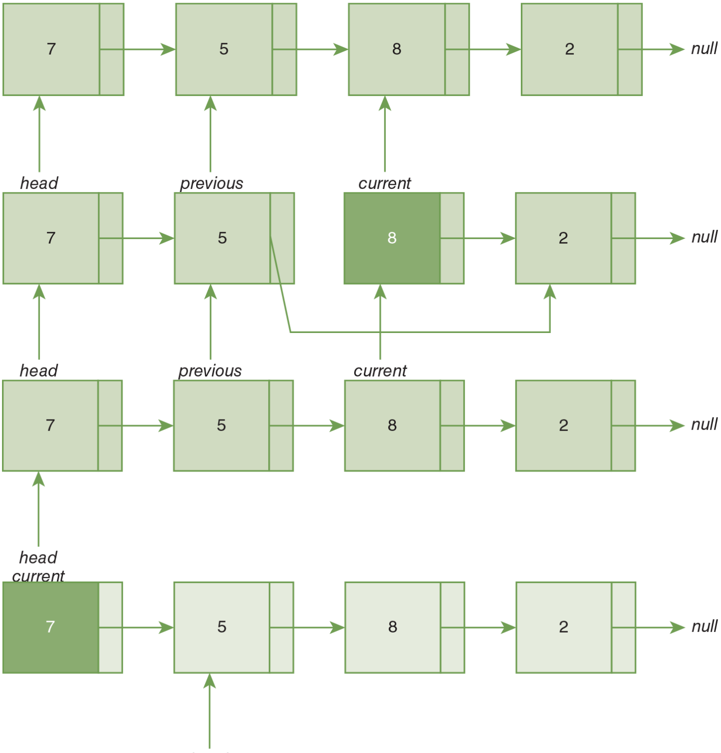
An *ArrayList* dynamically adjusts its size by increasing its capacity whenever it runs out of space. In fact, in many cases, the capacity exceeds the space needed to hold our elements. Then memory is allocated but not used. Obviously, this is not an efficient use of memory space.

However, *ArrayLists* are useful in some situations, such as reading data from a file, where we don’t know in advance how many items we will need to store in memory. Once the data is read, we know that the size of the *ArrayList* will not change further, so we can trim the capacity of the *ArrayList* to its current size using the *trimToSize* method, thus releasing the unused memory.

In other situations, however, the number of data items may dynamically increase or decrease as the program executes. For these cases, we need a data structure that efficiently grows and shrinks as items are added and removed.

A new data structure that we will illustrate in this chapter is the **linked list**, which can expand (or shrink) one object at a time, keeping the size of the list to a minimum at all times. An advantage, then, of linked lists is that they do not consume unnecessary memory.


5
7
Sarah
Mario
5
Ajay
Sonic
8
Gino
Diablo
2
Jin
Golf
null
7
5
8
2
null
head
7
5
8
2
null
head
6
null
6
7
5
8
2
null
head
6
7
5
8
2
null
head


7
5
8
2
null
head
previous
current
7
5
8
2
null
head
previous
current
7
5
8
2
null
head
current
7
5
8
2
null
head

```
Number of items in the list: 0

Number of items in the list: 1
7
Number of items in the list: 2
2 7
Number of items in the list: 3
5 2 7
8 could not be deleted:
2 was successfully deleted:
Number of items in the list: 2
5 7
7 was successfully deleted:
Number of items in the list: 1
5
5 was successfully deleted:
Number of items in the list: 0

8 could not be deleted:
Number of items in the list: 0
```

13.1 Linked Lists

13.1.1 Linked-List Concepts and Structure

A **linked list** can be thought of as a chain of linked nodes.

A **node** is an object with two attributes:

- data—The data can be a primitive data type (for example, an *int*), or it can be a reference to an object of a specified class.
- the location of the next node in the chain—We say that a node “**points to**” or “**refers to**” the next node.

Figure 13.1 shows how we can visualize a node containing the integer value 5. The arrow points to the next node in the list.

Figure 13.1 A Node

Description

In the last node of the list, the location of the next node contains the value *null*, to indicate that there are no more nodes in the list.

Figure 13.2 illustrates a linked list of four video game players.

The object data stored at each node has the following attributes: the player's ID, the player's name, and the name of the player's favorite game.

Figure 13.2 A Linked List

Description

From the standpoint of program design, this linked list can be implemented using three classes:

- a *Player* class, encapsulating a player
- a *PlayerNode* class, encapsulating a node
- a *PlayerLinkedList* class, encapsulating the linked list

In the *Player* class, we will have three instance variables:

- an *int* storing the user ID of the player
- a *String* storing the name of the player
- a *String* storing the name of the player's favorite game

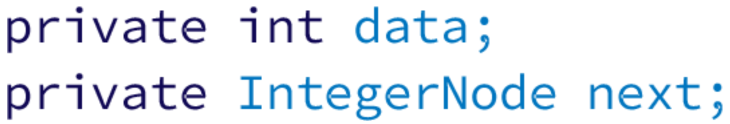
Often, a node class is designed in a general manner to store a generic *Object*. Implementing a list of generic *Objects* has the advantage of reusability; indeed, we could instantiate the list with any type of *Object* we want. In this chapter, we will first implement a linked list of the primitive type, *int*, then a linked list of *Player* objects, and then a linked list of generic *Objects*.

In the *IntegerNode* class, we have two instance variables:

- an *int*
- an *IntegerNode* object reference, representing the next node

Thus, the *IntegerNode* class is defined using an object reference of its own type. Indeed, one of its instance variables is an *IntegerNode* object reference.

We define our two instance variables using the following statements:



```
private int data;
private IntegerNode next;
```

Description

Based on this definition of the *IntegerNode* class, we need only one instance variable in the linked-list class, a reference to the first node, which we call the **head** of the linked list. Indeed, the first node will give us access to the second node, which in turn will give us access to the third node, and so on, until we reach the last node. We will know when we have reached the end of the linked list, because the reference to the next *IntegerNode* will have the value *null*.

Often, linked-list classes have another instance variable that holds the number of items in the linked list. Although the number of items can be calculated by looping through and counting all the nodes in the linked list, it is convenient to store the number of items as an instance variable. So our *IntegerLinkedList* class, encapsulating the linked list, will have two instance variables:

- an *IntegerNode* object reference, named *head*, representing the first node of the linked list
- an *int*, named *numberOfItems*, representing the number of items in the linked list

SOFTWARE ENGINEERING TIP

Include an instance variable in the linked-list class to store the number of items in the list for quick and direct access to that information as needed.

13.1.2 Linked-List Basics

Example 13.1 shows our *IntegerNode* class:

Example 13.1 The *IntegerNode* Class

```java
/* The IntegerNode class
   Anderson, Franceschi
*/

public class IntegerNode
{
  private int data;
  private IntegerNode next;

  /** default constructor
  *   sets data to 0, and next to null
  */
  public IntegerNode( )
  {
    data = 0;
    next = null;
  }

  /** overloaded constructor
  *   @param data data value
  */
  public IntegerNode( int data )
  {
    setData ( data );
    next = null;
  }

  /** accessor for data
  *   @return    the value of the node
  */
  public int getData( )
  {
    return data;
  }

  /** accessor for next
  *   @return   the reference to the next node
  */
  public IntegerNode getNext( )
  {
    return next;
  }

  /** mutator for data
  *   @param    data   the new value for the node
  *   @return  a reference to this object
  */
  public IntegerNode setData( int data )
  {
    this.data = data;
    return this;
  }

  /** mutator for next
  *   @param next the new value for next
  *   @return a reference to this object
  */
  public IntegerNode setNext( IntegerNode next )
  {
    this.next = next;
    return this;
  }
}
```

Description

The code for this class is straightforward. We code two constructors at lines 10 to 26. Both of these constructors set the value of *next* to *null*. This will be the desired action when a node is created. However, to allow a client (which will be the linked-list class) to reset the value of *next* as the list expands and shrinks, we provide the *setNext* method.

13.1.3 Methods of a Linked List

For our class encapsulating a linked list, we need to consider the following issues:

- We do not want client programs to change the head node of our list. Thus, we will not provide an accessor or a mutator for the head node.
- Client programs should not be able to change the number of items in the list. Only the methods of the class should update the number of items as we insert or delete items in the list. Thus, we will provide an accessor for the number of items in the list so that the client can view the number of items, but no mutator.

With a linked list, there is some basic functionality that we need to provide, such as

- insert an item,
- delete an item, and
- list, in order, all the items in the list, and return that list as a *String*.

Table 13.1 shows the APIs of the *insert*, *delete*, and *toString* methods.

IntegerLinkedList Methods

Methods of the *IntegerLinkedList* Class	
Return value	**Method name and argument list**

`void`	`insert( int value )` inserts *value* at the beginning of the list.
`boolean`	`delete( int value )` removes the first item on the list that is equal to *value* and returns *true*. If there is no such item on the list, the method returns *false*.
`String`	`toString( )` returns a *String* representation of the list.

Description

In our linked list of *ints*, we do not store the *ints* in any predetermined order. Thus, there are only two logical places to insert a node: at the beginning and at the end of the list. Inserting at the end will consume CPU time, since we will have to loop through all nodes in the list to find the end. So we have decided to insert at the beginning of the linked list because it is easier and faster. Unless we run out of memory, it will always be possible to insert a new node at the beginning of a list, so our *insert* method has a *void* return value.

Other options for implementing a linked list include providing methods to insert at the end of the list, or at a specified position in the list, or at a position before or after a node containing a specified value or object.

When inserting a new *int*, our *insert* method performs the following steps:

1. Instantiate a new node containing the *int* to be inserted.
2. Attach that node at the beginning of the list; that is, make that node point to the previous head node. If the list originally was empty, that is, *head* has the value *null*, then the *next* field of the new node is given the value *null*.
3. Indicate that the new node is now the head of the list; that is, make *head* point to the new node.
4. Increase the number of items in the list by 1.

Figures 13.3a to **13.3d** illustrate the first three steps.

Figure 13.3a Inserting: Our Original Linked List

Description

Figure 13.3b Step 1: The New Node Is Instantiated

Description

Figure 13.3c Step 2: The New Node Has Been Attached to the Beginning of the List

Description

Figure 13.3d Step 3: *head* Now Points to the New Node

Description

There are many alternatives for deleting an item from a linked list. We can delete the first element or the last item, or delete an item based on specified criteria. Such a criterion can be the value of the item, or it can be the position of an item in the list. We will implement one *delete* method only: one that deletes an item based on its value.

In order to delete an item, we will traverse the list searching for an item whose value matches a specified value passed as the argument of the *delete* method. **Traversing** a list means looping through the nodes in the list, one after the other, starting at the first node. If we find such an item, we will remove it from the list and return *true*. If we do not find the item, we will return *false*.

There are three possible outcomes when searching for such an item:

1. Such an item can be found and is located somewhere after the head node.
2. Such an item can be found and is located at the head node. Special care must be taken here. There will be a change in the head node of the list and our code will need

to handle that.

3. Such an item cannot be found. In this case, no deletion can take place, and we will return *false*.

In the first case, when the node to delete is located after the first node in the list, we need to connect the node before the deleted node (the “previous” node) to the node after the deleted node. To do this, we replace the previous node’s *next* field with the *next* field of the deleted node. Thus, as we traverse the list, we need to keep track of the previous node, as well as the current node. To do this, we maintain two node references, *previous* and *current*.

Once we have located the node to delete and it is not the first node of the list, we perform the following steps:

1. Set the *next* field in the *previous* node to the *next* field in the node to be deleted (*current*).
2. Decrease the number of items in the list by 1.

The *current* node becomes unreachable and is therefore a candidate for garbage collection.

Figures 13.4a and **13.4b** illustrate deleting a node with the value 8, which is located somewhere in the middle of the list.

Figure 13.4a The List Before Deleting the Item Whose Value Is 8

Description

Figure 13.4b The *previous* Node Is Connected to the Node After *current*, Deleting the Item Whose Value Is 8

Description

When the node to delete is the head node, we need to make the node pointed to by the deleted node the new *head* of the list. Thus, we perform the following steps:

1. Assign the *next* field of the *current* node to *head*.

2. Decrease the number of items in the list by 1.

Figures 13.5a and **13.5b** illustrate deleting the first node in the list.

Figure 13.5a Before Deleting the Item Whose Value Is 7

Description

Figure 13.5b After Updating *head*, the First Node in the List Is Deleted

Description

Example 13.2 shows our *IntegerLinkedList* class.

Example 13.2 The *IntegerLinkedList* Class

```java
/* The IntegerLinkedList class
   Anderson, Franceschi
*/

public class IntegerLinkedList
{
  private IntegerNode head;
  private int numberOfItems;

  /** default constructor
  *   constructs an empty list
  */
  public IntegerLinkedList( )
  {
    head = null;
    numberOfItems = 0;
  }

  /** accessor for numberOfItems
  *   @return   numberOfItems
  */
  public int getNumberOfItems( )
  {
    return numberOfItems;
  }

  /** insert method
  *   @param  value   data to insert
  *    inserts node at head
  */
  public void insert( int value )
  {
    IntegerNode nd = new IntegerNode( value );
    nd.setNext( head );
    head = nd;
    numberOfItems++;
  }

  /** delete method
  *   @param    value    the value to delete
  *   @return   true if value was deleted from the list, false otherwise
  */
  public boolean delete( int value )
  {
    IntegerNode current = head;
    IntegerNode previous = null;
    while ( current != null
           && current.getData( ) != value )
    {
      previous = current;
      current = current.getNext( );
    }

    if ( current == null ) // not found
         return false;
    else
    {
      if ( current == head )
        head = head.getNext( ); // delete head
      else
        previous.setNext( current.getNext( ) );

      numberOfItems--;
      return true;
    }
  }

  /** toString
  *   @return    values in list separated by a space
  */
  @Override
  public String toString( )
  {
    String listString = "";
    IntegerNode current = head;
    for ( int i = 0; i < numberOfItems; i++ )
    {
      listString += current.getData( ) + " ";
      current = current.getNext( );
    }
    return listString;
  }
}
```

Description

The default constructor (lines 10–17) initializes the *head* and *numberOfItems* instance variables to *null* and 0, respectively. The *insert* method, coded from lines 27 to 37, inserts a node containing its parameter *value* at the beginning of the list. At line 33, we create a new node, *nd*, with the data *value*. At line 34, we connect *nd* to the first node in the list by setting its *next* field to the current head of the list. At line 35, we assign the new node, *nd*, to *head*, making it the first node in the linked list. Figures 13.3a to 13.3d illustrate the impact on the list of lines 33–35. At line 36, we increment *numberOfItems* to reflect the addition of a node to the list.

The *delete* method, coded from lines 39 to 66, returns *true* if the deletion is successful and *false* if the deletion is not successful.

Using a *while* loop at lines 47–52, we walk through, or traverse, the list searching for a node containing *value*, our *delete* method's parameter. At lines 45 and 46, we declare and initialize two *IntegerNode* references, which we will use to track the current and previous nodes as we traverse the list. Because each node points only to the next node in the list, we can traverse the list in a forward direction only. Once we have reached a node, we do not have a way to backtrack to the previous node. Thus, as we traverse the list, we must remember the previous node because we will need to change the value of its next node. We update *previous* and *current* at lines 50–51 by assigning *current* to *previous*, then moving *current* to the next node in the list by calling the *getNext* method.

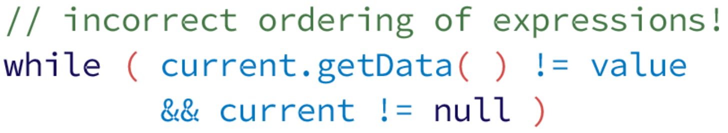
Once we find *value*, we will connect *previous* to the *IntegerNode* after *current*. If we have reached the end of the list —that is, *current* is *null* (line 47)—or if we have found *value* (line 48), we are ready to either return *false* or delete the node by updating the links in our list. At that point, we exit the *while* loop and skip to line 54.

COMMON ERROR TRAP

When traversing a list, always test if a node reference is *null* before calling a method using that reference. Failure to do so might result in a *NullPointerException* at run time. More generally, when coding a linked-list method, always pay attention to the possibility of an object reference being *null* before using that reference to call a method.

Note that the order of the expressions in the *while* loop condition is critical. Expressions in a compound condition are evaluated left to right, so (*current != null*) is evaluated first. If this expression is *false* (that is, *current* is *null*), then the whole *while* loop condition cannot evaluate to *true*, so the second expression is not evaluated. This is important because the second expression uses *current* to call the *getData* method. If *current* is *null*, the evaluation of the second expression would generate a *NullPointerException*. In this way, we are taking advantage of Java's short-circuit evaluation of logical AND operations.

Thus, if we reversed the order of the expressions in the *while* loop condition, as shown here,



```
// incorrect ordering of expressions!
while ( current.getData( ) != value
          && current != null )
```

Description

reaching the end of the list would always generate a *NullPointerException*.

At line 54, we test whether *current* is *null*, because a *null* value indicates that we exited the *while* loop because the list is empty or we reached the end of the list without finding *value*. Either way, the deletion is unsuccessful and we return *false* at line 55.

If *current* is not *null*, we have found *value*. We update the list and return *true* at lines 58–64. If *current* is the head node, we found *value* at the beginning of the list, so we need to update the *head* instance variable. In this case, we assign the node after *head* to *head* at line 59. Note that if there was only one element in the list before the deletion, *head* becomes *null* (at this point the list will be empty). Figures 13.5a and 13.5b show the impact of executing line 59 on the list (before and after).

If *current* is not the head node, we skip to line 61 where we set the node pointed to by *previous* to be the node after *current*. At this point, *current* is no longer part of the linked list.

At line 63, we decrement *numberOfItems*, to reflect that we now have one fewer item in the list. Finally, at line 64, we return *true*. Figures 13.4a and 13.4b show the impact of executing line 61 on the list (before and after).

Our last method, *toString*, is at lines 68–82. Our *toString* method traverses the list and returns a *String* containing the data from each item in the list. A *toString* method traversing the list is especially useful at the debugging stage, when we want to verify that we have properly added or deleted an element. We can test our code by calling *toString* before and after such operations.

13.1.4 Testing a Linked-List Class

Like any class that we design, we want to test the class before using it in a program. In particular, we should test two important methods: *insert* and *delete*. Furthermore, we want to test all possible scenarios.

SOFTWARE ENGINEERING TIP

There are many ways to code the deletion of a node in the list. Try to write code that is easy to read and maintain.

Considering our *insert* method, which always inserts at the head of the list, we want to test a minimum of two situations:

- inserting into an empty list
- inserting into a nonempty list

As we will see later in the chapter, there are other types of linked lists and their *insert* methods may require more test cases than the ones previously mentioned.

After each insertion, we can use the *toString* method to verify that the items were inserted correctly.

For the *delete* method, we should test the following scenarios:

- attempting to delete from an empty list
- deleting an item in the middle of the list
- deleting an item stored in the head node
- deleting an item stored in the last node in the list

- attempting to delete an item not in the list

After each deletion, we can use the *toString* method to check that the items were deleted correctly.

Example 13.3 shows a client program that tests the *IntegerLinkedList* class.

Example 13.3 The *IntegerLinkedListTest* Class

```
/* The IntegerLinkedListTest class
   Anderson, Franceschi
*/

public class IntegerLinkedListTest
{
  public static void main( String [ ] args )
  {
    // construct empty IntegerLinkedList
    IntegerLinkedList numbers = new IntegerLinkedList( );
    System.out.println( "Number of items in the list: "
     + numbers.getNumberOfItems( ) + "\n" + numbers.toString( ) );

    numbers.insert( 7 ); // insert in empty list
    System.out.println ( "Number of items in the list: "
     + numbers.getNumberOfItems( ) + "\n" + numbers.toString( ) );

    numbers.insert( 2 ); // insert in list with one item
    System.out.println( "Number of items in the list: "
     + numbers.getNumberOfItems ( ) + "\n" + numbers.toString( ) );

    numbers.insert( 5 ); // insert in list with two items
    System.out.println( "Number of items in the list: "
     + numbers.getNumberOfItems( ) + "\n" + numbers.toString( ) );

    if ( ! numbers.delete( 8 ) )     // unsuccessful - not in list
       System.out.println( "8 could not be deleted:" );

    if ( numbers.delete( 2 ) )       // successful
       System.out.println( "2 was successfully deleted:" );
    System.out.println( "Number of items in the list: "
       + numbers.getNumberOfItems( ) + "\n" + numbers.toString( ) );

    if ( numbers.delete( 7 ) )       // successful
       System.out.println( "7 was successfully deleted:" );
    System.out.println( "Number of items in the list: "
       + numbers.getNumberOfItems( ) + "\n" + numbers.toString( ) );

    if ( numbers.delete( 5 ) )       // successful
       System.out.println( "5 was successfully deleted:" );
    System.out.println( "Number of items in the list: "
       + numbers.getNumberOfItems( ) + "\n" + numbers.toString ( ) );

    if ( ! numbers.delete( 8 ) )     // unsuccessful - empty list
       System.out.println( "8 could not be deleted:" );
    System.out.println( "Number of items in the list: "
       + numbers.getNumberOfItems( ) + "\n" + numbers.toString ( ) );
  }
}
```

Description

In this example, we instantiate the *IntegerLinkedList numbers* object at line 10, then traverse the empty list at lines 11–12.

SOFTWARE ENGINEERING TIP

Testing all the methods in a linked list is critical to avoid errors at run time. Try to test all possible scenarios of all methods.

We successively insert 7, 2, and 5 and traverse *numbers* after each insertion at lines 14–24.

After that, we test our *delete* method. At line 26, we attempt to delete the value 8; we know this will fail, as the output in **Figure 13.6** shows.

Figure 13.6 Output of Example 13.3

Description

We then delete in the middle of the list at line 29, at the end of the list at line 34, and at the beginning of the list (actually the only item left at that point) at line 39. Another attempt to delete is made at line 44, but at that time the list is empty, which causes us to execute line 45, as shown in Figure 13.6.

CODE IN ACTION

Within the online resources, you will find a movie with a step-by-step illustration of linked-list methods. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

```
Number of items in the list: 0

Number of items in the list: 1
id: 7          name: Sarah     game: Mario

Number of items in the list: 2
id: 2          name: Jin       game: Golf
id: 7          name: Sarah     game: Mario

Number of items in the list: 3
id: 5          name: Ajay      game: Sonic
id: 2          name: Jin       game: Golf
id: 7          name: Sarah     game: Mario

8 not found: cannot be deleted

Player retrieved: id: 2 name: Jin              game: Golf
Number of items in the list: 3
id: 5          name: Ajay      game: Sonic


id: 2          name: Jin       game: Golf
id: 7          name: Sarah     game: Mario

Player deleted: id: 2 name: Jin                game: Golf
Number of items in the list: 2
id: 5          name: Ajay      game: Sonic
id: 7          name: Sarah     game: Mario

Player deleted: id: 7          name: Sarah     game: Mario
Number of items in the list: 1
id: 5          name: Ajay      game: Sonic

Player deleted: id: 5          name: Ajay      game: Sonic
Number of items in the list: 0

7 not found: cannot be deleted
```

13.2 Linked Lists of Objects

Our next step is to design and code a linked list of objects, for example, *Player* objects, described earlier in the chapter.

Since each node in our list will store a *Player* reference, we start by defining our *Player* class, shown in **Example 13.4**.

Example 13.4 The *Player* Class

```java
/* The Player Class
   Anderson, Franceschi
*/

public class Player
{
 private int id;
 private String name;
 private String game;

 /** constructor
 * @param id    player's id
 * @param name  player's name
 * @param game  player's game
 */
 public Player( int id, String name, String game )
 {
    setID( id );
    setName( name );
    setGame( game );
 }

 /** accessor for id
 * @return   id
 */
 public int getID( )
 {
    return id;
 }

 /** accessor for name
 * @return name
 */
 public String getName( )
 {
    return name;
 }

 /** accessor for game
 * @return game
 */
 public String getGame( )
 {
    return game;
 }

 /** mutator for Id
 * @param id  new value for id
 * @return    a reference to this object
 */
 public Player setID( int id )
 {
   this.id = id;
   return this;
 }

 /** mutator for name
 * @param name  new value for name
 * @return      a reference to this object
 */
 public Player setName( String name )
 {
    this.name = name;
    return this;
 }

 /** mutator for game
 * @param game  new value for game
 * @return      a reference to this object
 */
 public Player setGame( String game )
 {
    this.game = game;
    return this;
 }

 /** equals method
 *  @param o reference to object to compare to this object
 *  @return  true if o is a Player object
 *   and id, name, and game are equal in both objects; false otherwise
 */
 @Override
 public boolean equals( Object o )
 {
   if (  ! ( o instanceof  Player ) )
       return false;
   else
   {
       Player  objPlayer = ( Player ) o;
       return ( id == objPlayer.id && name.equals( objPlayer.name )
              && game.equals( objPlayer.game ) );
   }
 }

 /** toString method
 *  @return  String representation of Player object
 */
 @Override
 public String toString( )
 {
    return ( "id: " + id + "\tname: "
             + name + "\tgame: " + game );
 }
}
```

Description

The code for this class is straightforward. We declared the three instance variables, along with a constructor, accessors and mutators, and the standard *equals* and *toString* methods.

Example 13.5 shows our *PlayerNode* class:

Example 13.5 The *PlayerNode* Class

```java
/* The PlayerNode class
   Anderson, Franceschi
*/

public class PlayerNode
{
 private Player player;
 private PlayerNode next;
 /** default constructor
 * initializes player and next references to null
 */
 public PlayerNode( )
 {
    player = null;
    next = null;
  }

  /** overloaded constructor
  * @param  player
  * initializes player reference to player reference to Player object
  */
  public PlayerNode( Player player )
  {
    setPlayer( player );
    next = null;
  }

  /** accessor for player
  * @return player
  */
  public Player getPlayer( )
  {
    return player;
  }

  /** accessor for next
  * @return next
  */
  public PlayerNode getNext( )
  {
    return next;
  }

  /**  mutator for player
  *  @param player  new Player reference
  *  @return        a reference to this object
  */
  public PlayerNode setPlayer( Player player )
  {
    this.player = player;
    return this;
  }

  /**  mutator for next
  *  @param next  new reference to next PlayerNode
  *  @return      a reference to this object
  */
  public PlayerNode setNext( PlayerNode next )
  {
    this.next = next;
    return this;
  }
}
```

Description

The code for this class is similar to the code of the *IntegerNode* class. The overloaded constructor allows the client to set the *Player* object, while the default constructor sets the reference for the *Player* object to *null*.

13.2.1 A Linked-List Shell

For our class encapsulating a linked list of *Player* objects, we need to consider the following issues:

- We anticipate having many linked-list classes. Therefore, it makes sense to set up a linked-list superclass from which our more specialized linked-list classes will inherit.
- We provide some basic utility methods, but we omit methods to insert or delete nodes in the list, because those methods will have different names and implementations, depending on the functionality of a given subclass.
- We do not intend to instantiate objects from our superclass; thus, we declare our superclass *abstract*.

Example 13.6 shows our *abstract ShellLinkedList* class. This class defines methods that will be common to all subclasses. For example, in addition to the default constructor and the accessor for the number of items in the list, we provide a method to determine whether the list is empty and a *toString* method that can be used to print each node in the list. We declare both instance variables as *protected* so that our linked-list subclasses inherit the head and number of items in the list.

Example 13.6 The *ShellLinkedList* Class

```java
/* The ShellLinkedList class
   Anderson, Franceschi
*/

public abstract class ShellLinkedList
{
  protected PlayerNode head;
  protected int numberOfItems;

  /**  constructor
  *  sets head to null and numberOfItems to 0
  */
  public ShellLinkedList( )
  {
    head = null;
    numberOfItems = 0;
  }

  /**  accessor for numberOfItems
  *   @return    numberOfItems
  */
  public int getNumberOfItems( )
  {
    return numberOfItems;
  }

  /** isEmpty method
  * @return  true if no items in list; false otherwise
  */
  public boolean isEmpty( )
  {
    return ( numberOfItems == 0 );
  }

  /** toString method
  * @return the contents of the list
  */
  @Override
  public String toString( )
  {
    String listString = "";
    PlayerNode current = head;
    while ( current != null )
    {
      listString += current.getPlayer( ).toString( ) + "\n";
      current = current.getNext( );
    }
    return listString;
  }
}
```

Description

SOFTWARE ENGINEERING TIP

Do not include a mutator method for the number of items. Only the linked-list class should alter the number of items as items are inserted or deleted. Including a mutator method for the number of items could allow the client to corrupt its value.

Table 13.2 shows the APIs of *ShellLinkedList* constructor and methods.

***ShellLinkedList* Constructor and Methods**

Constructor and Methods of the *abstract ShellLinkedList* Class

Class	**Constructor and argument list**
`ShellLinkedList`	`ShellLinkedList( )` constructs an empty list
Return value	**Method name and argument list**
`int`	`getNumberOfItems( )` returns the number of items in the list
`boolean`	`isEmpty( )` returns *true* if the list contains 0 items, *false* otherwise
`String`	`toString( )` returns the contents of every node in the list

Description

SOFTWARE ENGINEERING TIP

Do not provide an accessor or mutator for the head node instance variable of the linked list. This will protect the head node from being accessed or changed outside the class.

SOFTWARE ENGINEERING TIP

Choose names for instance variables and methods that illustrate their function within the data structure. Your class will be easier for others and yourself to understand at maintenance time.

SOFTWARE ENGINEERING TIP

Provide a *toString* method that traverses the list. This is helpful in testing the other methods of the class. In particular, traversing the list after calling the insert or delete methods can verify that an item was correctly added or removed.

We coded the method *isEmpty* at lines 27 to 33. It returns *true* if the list is empty, *false* otherwise.

The *toString* method, at lines 35–49, traverses the linked list until it reaches the end of the list, where *current* will be *null*.

13.2.2 Generating an Exception

Now that we have a shell class for a linked list, we want to add some methods to perform operations on the linked list, such as inserting or deleting elements.

An issue may arise with the return value of the *delete* method. When deleting a node, we want to return the item that we are deleting. Indeed, we want to be able to delete a node based on the value of one or more of the fields of the object stored at that node, that is, the value of one or more of the instance variables of the *Player* class. For example, if the client wants to delete the first *Player* on the list with an ID of 5, we would then return that *Player* object to the client. If the list is empty or we cannot find a *Player* with ID 5, we do not want to return *null* because the client likely will attempt to use the returned object reference, which would generate a *NullPointerException*. A solution to this problem is to *throw* an exception when we are unable to delete the requested node. To do this, we create our own exception class, *DataStructureException*, which we use throughout this chapter.

It is good practice to define our own exception class as a subclass of another exception class. This way, our class inherits the existing functionality of the exception class, which simplifies coding the new class: we need to code only the constructor.

Example 13.7 shows our *DataStructureException* class, which *extends* the *NoSuchElementException* class, which is in the *java.util* package.

Example 13.7 The *DataStructureException* Class

```
 1 /* The DataStructureException Class
 2    Anderson, Franceschi
 3 */
 4 import java.util.NoSuchElementException;
 5
 6 public class DataStructureException extends NoSuchElementException
 7 {
 8   /** constructor
 9   * @param s error message
10   */
11   public DataStructureException( String s )
12   {
13      super( s );
14   }
15 }
```

Description

The constructor for the class is coded at lines 8 to 14; it simply takes a *String* parameter and passes it to the superclass constructor. When one of our methods detects an error situation, such as an attempt to delete from an empty list, we will *throw* the exception using a statement like the following:

```
throw new DataStructureException( "Some error message here" );
```

Description

The message we pass to the constructor will identify the type of error we detected.

The header of any method that *throws* the *DataStructureException* will add a *throws* clause, as in the following template:

```
accessModifier dataType methodName( parameter list )
                              throws DataStructureException
```

Description

The *NoSuchElementException* class inherits from *RuntimeException*, and thus is not a checked exception. This means that the client does not need to use *try/catch* blocks when invoking any method that could throw a *DataStructure-*

Exception. Nevertheless, we will always use *try/catch* blocks so that when a *DataStructureException* occurs, the client will see our custom message and can better understand the source of the problem.

Now we are ready to expand our shell linked-list class with more meaningful methods.

13.2.3 Other Methods of a Linked List

In addition to our *insert* and *delete* methods, we also provide the functionality to retrieve, or **peek** at, the contents of a node, without deleting it.

Table 13.3 shows the APIs of the *insert*, *delete*, and *peek* methods.

PlayerLinkedList Methods

Methods of the *PlayerLinkedList* Class

Return value	Method name and argument list
`void`	`insert( Player p )` inserts *Player p* at the beginning of the list.
`Player`	`delete( int searchID )` returns and removes the first *Player* in the list with an ID equal to *searchID*. If there is no such *Player* in the list, the method throws a *DataStructureException*.
`Player`	`peek( int searchID )` returns the first *Player* on the list whose ID is equal to *searchID*. If there is no such *Player* in the list, the method throws a *DataStructureException*.

Description

Again, we will insert at the beginning of the list and our *insert* method has a *void* return value; our insert method works in the same manner as the insert method for a list of *ints*.

Our *delete* method will also delete a node based on specific criteria. With our list of *ints*, the criterion was simply the value of an item; here, the criterion can be the value of one (or several) instance variables of an item. We will implement one *delete* method only: one that deletes an item based on a specified value of its *id* field. The implementation of a *delete* method that deletes an item based on a specified value of its *name* or *game* field is similar.

If we find such an item, we will remove it from the list and return a reference to that item. When deleting from a list of *ints*, it made sense to return a *boolean* value rather than the value of the item, because we already knew the value of the item since it was passed as a parameter to the method. Here, we delete a node based on the value of the *id* field of an object, so we do not know the values of the other fields of that object. Thus, it makes sense to return an object reference rather than just *true* or *false*. If the item cannot be found, we will *throw* a *DataStructureException*.

Otherwise, the mechanics of deleting a node within the list, whether that node is the *head* node or is in the middle of the list, are exactly the same as with our linked list of *ints*.

Example 13.8 shows our *PlayerLinkedList* class. This class *extends* and inherits the functionality of our *ShellLinkedList* class; thus *head* and *numberOfItems* are inherited instance variables.

Example 13.8 The *PlayerLinkedList* Class

```java
/* The PlayerLinkedList class
   Anderson, Franceschi
*/

public class PlayerLinkedList extends ShellLinkedList
{

  /** default constructor
  *   calls constructor of ShellLinkedList class
  */
  public PlayerLinkedList( )
  {
    super( );
  }

  /** insert method
  *   @param      p    Player object to insert
  */
  public void insert( Player p )
  {
    // insert as head
    PlayerNode pn = new PlayerNode( p );
    pn.setNext( head );
    head = pn;
    numberOfItems++;
  }

  /** delete method
  *   @param      searchID    id of Player to delete
  *   @return    the Player deleted
  */
  public Player delete( int searchID )
                           throws DataStructureException
  {
   PlayerNode current = head;
   PlayerNode previous = null;
   while ( current != null
           && current.getPlayer( ).getID( ) != searchID )
   {
     previous = current;
     current = current.getNext( );
   }

   if ( current == null ) // not found
        throw new DataStructureException( searchID
                    + " not found: cannot be deleted" );
   else
   {
     if ( current == head )
       head = head.getNext( ); // delete head
     else
       previous.setNext( current.getNext( ) );

     numberOfItems--;
     return current.getPlayer( );
   }
  }

  /** peek method
  *   @param     searchID   id of Player to search for
  *   @return    the Player found
  */
  public Player peek( int searchID )
               throws DataStructureException
  {
    PlayerNode current = head;
    while ( current != null
            && current.getPlayer( ).getID( ) != searchID )
    {
      current = current.getNext( );
    }

    if ( current == null ) // not found
      throw new DataStructureException( searchID
                   + " not found: cannot be deleted" );
    else
    {
      return current.getPlayer( );
    }
 }
}
```

Description

The default constructor (lines 8–14) calls the constructor of the superclass to initialize the *head* and *numberOfItems* instance variables. The *insert* method, coded from lines 16 to 26, is similar to our insert method for a linked list of *ints*, except that it inserts a *Player* object instead of an *int*.

The *delete* method, coded from lines 28 to 57, returns the *Player* deleted if the deletion was successful and *throws* a *DataStructureException* if the deletion was not successful.

The *while* loop at lines 37–42 is very similar to the *delete* method for our linked list of *ints*. We first traverse the list searching for a node containing a *Player* object whose *id* has the same value as *searchID*, our *delete* method's parameter.

Once we find a *Player* whose *id* field matches *searchID*, we will connect *previous* to the *PlayerNode* after *current*. If we have reached the end of the list, that is, *current* is *null* (line 37), or if we have found a *Player* whose *id* value is *searchID* (line 38), we are ready to either *throw* an exception or delete the node by updating the links in our list. At that point, we exit the *while* loop and skip to line 44.

At line 44, we test whether *current* is *null*, because a *null* value indicates that we exited the *while* loop because the list is empty or we reached the end of the list (without finding a *Player* whose id is *searchID*). If the deletion is unsuccessful, we *throw* a *DataStructureException* with an appropriate message at lines 45–46.

If *current* is not *null*, that means that we have found a *Player* whose *id* is *searchID*. We update the list and return the deleted *Player* at lines 49–55.

The *peek* method is coded at lines 59–80. We traverse the list in the same way as the *delete* method, except that because we will not delete a node, we do not need to mark the node before *current*.

If we do not find a node containing a *Player* whose id is *searchID*, we *throw* an exception at lines 74–75. If we find one, we return that *Player* object. In this way, the client can directly update the objects in the list.

13.2.4 Testing a Linked-List Class

Again, we want to test the class before using it in a program, and we want to test all possible scenarios, similarly to what we did with our linked list of *ints*.

After each insertion and deletion, we use the *toString* method to verify that the items were inserted and deleted correctly.

Example 13.9 shows a client program that tests the *PlayerLinkedList* class.

Example 13.9 The *PlayerLinkedListTest* Class

```java
/* The PlayerLinkedListTest class
   Anderson, Franceschi
 */

public class PlayerLinkedListTest
{
  public static void main( String [ ] args )
  {
    Player p1 = new Player( 7,"Sarah","Mario" );
    Player p2 = new Player( 2,"Jin","Golf" );
    Player p3 = new Player( 5,"Ajay","Sonic" );

    // construct empty PlayerLinkedList
    PlayerLinkedList players = new PlayerLinkedList( );
    System.out.println( "Number of items in the list: "
     + players.getNumberOfItems( ) + "\n" + players.toString( ) );

    players.insert( p1 );  // insert in empty list
    System.out.println( "Number of items in the list: "
     + players.getNumberOfItems( ) + "\n" + players.toString( ) );

    players.insert( p2 );  // insert in list of one item
    System.out.println( "Number of items in the list: "
     + players.getNumberOfItems( ) + "\n" + players.toString ( ) );

    players.insert( p3 );  // insert in list of two items
    System.out.println( "Number of items in the list: "
     + players.getNumberOfItems( ) + "\n" + players.toString( ) );

    Player temp;    // will be assigned the deleted item

    try
    {
     temp = players.delete( 8 );           // unsuccessful
     System.out.println( "Player deleted: " + temp );
    }
    catch ( DataStructureException dse1 )
    {
     System.out.println( dse1.getMessage( ) + "\n" );
    }

    try
    {
     temp = players.peek( 2 );             // test peek
     System.out.println( "Player retrieved: " + temp );
     System.out.println( "Number of items in the list: "
       + players.getNumberOfItems( ) + "\n" + players.toString( ) );

     temp = players.delete( 2 );           // delete in the middle
     System.out.println( "Player deleted: " + temp );
     System.out.println( "Number of items in the list: "
       + players.getNumberOfItems( ) + "\n" + players.toString( ) );

     temp = players.delete( 7 );           // delete the last item
     System.out.println( "Player deleted: " + temp );
     System.out.println( "Number of items in the list: "
      + players.getNumberOfItems( ) + "\n" + players.toString( ) );

     temp = players.delete( 5 );         // delete the first item
     System.out.println( "Player deleted: " + temp );
     System.out.println( "Number of items in the list: "
      + players.getNumberOfItems( ) + "\n" + players.toString( ) );

     temp = players.delete( 7 );         // delete from empty list
     System.out.println( "Player deleted: " + temp );
     System.out.println( "Number of items in the list: "
      + players.getNumberOfItems( ) + "\n" + players.toString( ) );
    }
    catch ( DataStructureException dse2 )
    {
      System.out.println( dse2.getMessage( ) );
    }
  }
}
```

Description

In this example, we instantiate three *Player* object references *p1*, *p2*, and *p3* at lines 9, 10, and 11. We instantiate the *PlayerLinkedList* players object reference at line 14, then traverse the empty list *players* at lines 15–16.

We successively insert *p1*, *p2*, and *p3* and traverse *players* after each insertion at lines 18–28.

After that, we test our *delete* method. Because our *delete* method *throws* a *DataStructureException*, we use *try* and *catch* blocks when calling that method.

At line 34, we attempt to delete an item in the list whose *id* is 8; we know this will fail, and as the output shows in **Figure 13.7**, we execute the *catch* block at lines 37–40.

Figure 13.7 Output of Example 13.9

Description

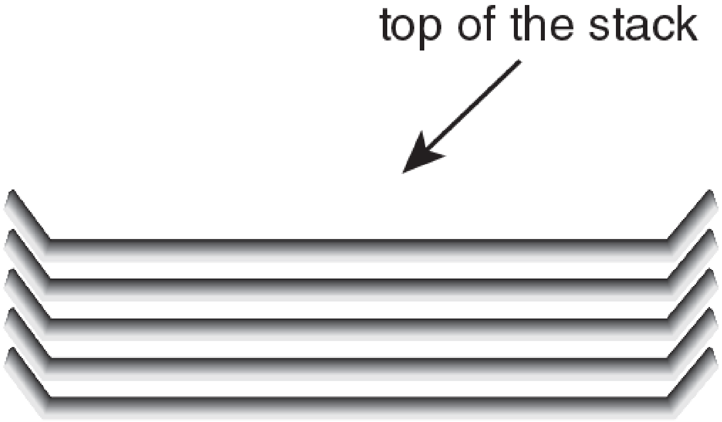
In the next *try* block, at line 44, we call the *peek* method to see if there is a *Player* whose *id* is 2. We traverse the list at lines 46–47 and can verify that the list has not been modified by the call to *peek*.

We then delete in the middle of the list at line 49, at the end of the list at line 54, and at the beginning of the list (actually the only item left at that point) at line 59. Another attempt to delete is made at line 64, but at that time the list is empty. This causes us to execute the second *catch* block at lines 69–72, as shown in **Figure 13.7**.

with these end-of-chapter questions

13.14.2	Reading and Understanding Code Questions 14, 15, 16, 17, 18, 19, 20, 21
13.14.3	Fill In the Code Questions 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
13.14.4	Identifying Errors in Code Questions 33, 34
13.14.5	Debugging Area—Using Messages from the Java Compiler and Java JVM

	Questions 39, 40, 41, 42
13.14.6	Write a Short Program Questions 43, 44, 45, 46, 48, 50
13.14.8	Technical Writing Question 72



13.3 Implementing a Stack Using a Linked List

Imagine a group of college students on a spring break, sharing an apartment. After they eat, they typically pile up the dirty dishes in the kitchen sink. Another meal is consumed, and more dirty dishes are piled on top of the existing ones. At the top of the pile is the dirty dish that was placed there last. Soon the students run out of clean dishes, and somebody will have to start cleaning them. He or she will start by cleaning the dish at the top of the pile, that is, the last dish placed on the pile. That approach is called **last in, first out**, or **LIFO**.

A **stack** is a linear data structure that organizes items in a last in, first out manner. **Figure 13.8** shows a stack of trays. The tray at the top of the stack was put on the stack last, but will be taken off the stack first.

Figure 13.8 A Stack of Trays

Description

A stack can be represented by a linked list. In a linked list representing a stack:

- we insert, or **push**, at the beginning of the list
- we delete, or **pop**, the item at the beginning of the list

Since we insert and delete at the beginning of the list, the item deleted is the last one that was inserted, reflecting the LIFO pattern.

Table 13.4 shows the APIs of the *push*, *pop*, and *peek* methods.

PlayerStackLinkedList Methods

Methods of the *PlayerStackLinkedList* Class

Return value	**Method name and argument list**
`void`	`push( Player p )` inserts *Player p* at the top of the stack.
`Player`	`pop( )` returns and removes the first *Player* of the list. If the list is empty, the method throws a *DataStructureException*.
`Player`	`peek( )` returns the first *Player* on the list without deleting it. If the list is empty, the method throws a *DataStructureException*.

Description

The *push* method is identical to the *insert* method of the *PlayerLinkedList* class discussed earlier, and is illustrated in Figures 13.3a to 13.3d.

The *pop* method is different from the *delete* method we coded earlier in our *PlayerLinkedList* class. In a stack, we always delete the first item in the list. Therefore, in a linked list implementing a stack, we do not delete an item based on the value of one of its instance variables. The *pop* method for our stack returns a *Player* object, the one stored at the head of the linked list. If our stack is empty, our *pop* method will *throw* a *DataStructureException* in order to avoid returning a *null* object reference.

The steps required to pop the first item are identical to the steps for deleting the first node in the *PlayerLinkedList*. That is illustrated in Figures 13.5a and 13.5b.

Example 13.10 shows our *PlayerStackLinkedList* class. This class also *extends* and inherits the functionality of our *ShellLinkedList* class.

Example 13.10 The *PlayerStackLinkedList* Class

```java
/* The PlayerStackLinkedList class
   Anderson, Franceschi
*/

public class PlayerStackLinkedList extends ShellLinkedList
{
   // head and numberOfItems are inherited instance variabl

   public PlayerStackLinkedList( )
   {
      super( );
   }

   /** push method
   *   @param p Player object to insert
   */
   public void push( Player p )
   {
     PlayerNode pn = new PlayerNode( p );
     pn.setNext( head );
     head = pn;
     numberOfItems++;
   }

   /** pop method
   *   @return    the Player object deleted
   */
   public Player pop( ) throws DataStructureException
   {
     if ( isEmpty( ) )
       throw new DataStructureException
                  ( "empty stack: cannot be popped" );
     else
     {
       Player deleted = head.getPlayer( );
       head = head.getNext( );
       numberOfItems--;
       return deleted;
     }
   }

   /** peek method
   *   @return    the Player object retrieved
   */
   public Player peek( ) throws DataStructureException
   {
     if ( isEmpty( ) )
          throw new DataStructureException
                     ( "empty stack: cannot peek" );
     else
     {
          return head.getPlayer( );
     }
   }
}
```

Description

The *push* method, coded from lines 14 to 23, is identical to the *insert* method in the *PlayerLinkedList* class (Example 13.8).

In the *pop* method (lines 25–40), we first test if the stack is empty at line 30. If it is empty, we *throw* a *DataStructureException* with the appropriate argument at lines 31–32. If the stack is not empty, we delete the first item in the stack and return it. We call the *getPlayer* method from the *PlayerNode* class to get the *Player* stored at the head of the stack, and assign it to the *Player* reference *deleted* (line 35). The *deleted* reference is then returned at line 38. At lines 36 and 37, we perform the bookkeeping on the stack to reflect the deletion. We update *head* at line 36 and decrement *numberOfItems* at line 37.

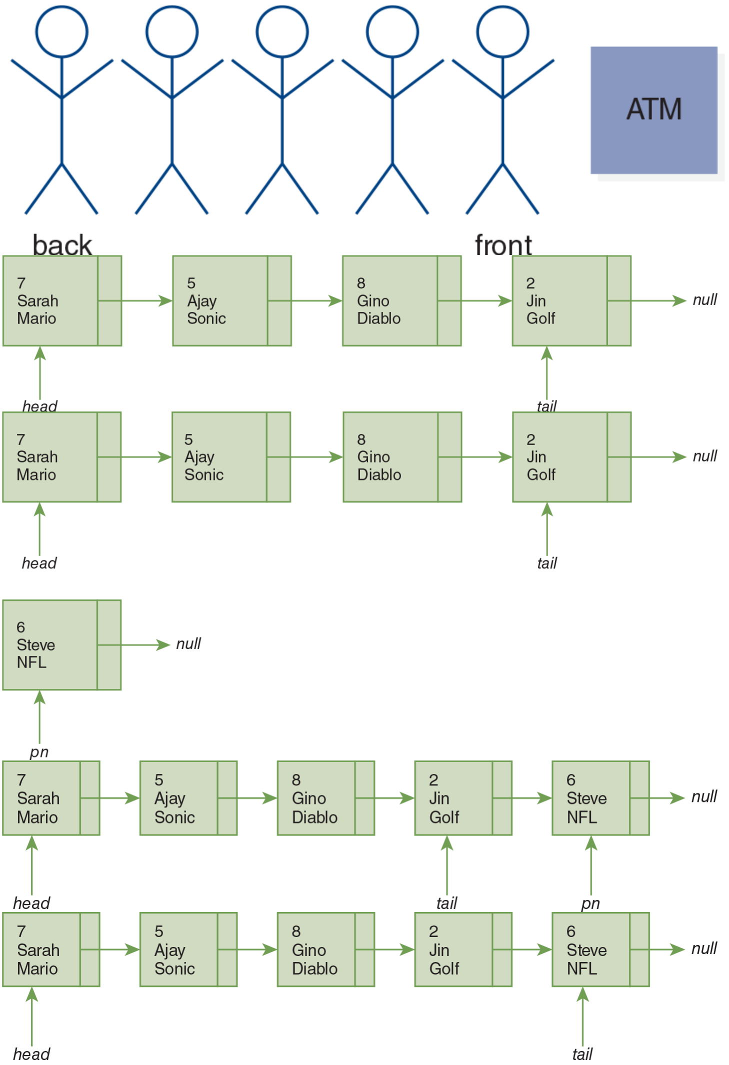
COMMON ERROR TRAP

Before popping an item from a linked list representing a stack, always check if the linked list is empty. Not doing so results in a *NullPointerException* at run time.

The *peek* method is coded at lines 42–54. If the list is empty, we *throw* an exception at lines 48–49. If the list is not empty, we return the *Player* at the head of the list at line 52.

Like our previous linked-list implementation, it is very important to test if the stack is empty before trying to delete a node; failure to do so will generate a *NullPointerException* at run time.

A similar program to Example 13.9 can be coded to test all possible scenarios when using the methods of the *PlayerStackLinkedList* class. This is proposed in the short program section of the exercises.


ATM
back
front
7 Sarah Mario
5 Ajay Sonic
8 Gino Diablo
2 Jin Golf
null
head
tail
7 Sarah Mario
5 Ajay Sonic
8 Gino Diablo
2 Jin Golf
null
head
tail
6 Steve NFL
null
pn
7 Sarah Mario
5 Ajay Sonic
8 Gino Diablo
2 Jin Golf
6 Steve NFL
null
head
tail
pn
7 Sarah Mario
5 Ajay Sonic
8 Gino Diablo
2 Jin Golf
6 Steve NFL
null
head
tail

13.4 Implementing a Queue Using a Linked List

Imagine a line of people at an automated teller machine, or ATM, waiting to withdraw cash. The person at the front of the line is using the ATM. When a new customer arrives, the customer goes to the back of the line. As customers use the ATM, they exit the line, and the next customer moves to the front of the line. Thus, customers use the ATM in the order of their arrival times. We call this pattern “**first in, first out,**” or **FIFO**.

A **queue** is a linear data structure that organizes items in a first in, first out manner.

Figure 13.9 shows a queue of people at an ATM. The person at the front of the queue arrived first and will use the ATM first. The person at the back arrived last and will use the ATM last. The next person to arrive will stand after the person currently at the back of the queue. That newly arrived person will become the new back of the line.

Figure 13.9 A Queue of People Waiting at the ATM

Description

A queue can be represented by a linked list by providing the following operations:

- we insert, or **enqueue**, an item at the end of the list
- we delete, or **dequeue**, the item at the beginning of the list
- we *peek* at the item at the beginning of the list

Table 13.5 shows the APIs of the *enqueue*, *dequeue*, and *peek* methods.

PlayerQueueLinkedList Methods

Methods of the *PlayerQueueLinkedList* Class	
Return value	**Method name and argument list**
`void`	`enqueue( Player p )` inserts *Player p* at the end of the list.
`Player`	`dequeue( )` returns and removes the first *Player* from the list. If the list is empty, the method throws a *DataStructureException*.

`Player`	`peek( )` returns the first *Player* on the list, but does not delete the *Player*. If the list is empty, the method throws a *DataStructureException*.

Description

We can implement a queue using a linked list; however, we will make an important change. Because a queue inserts items at the end of the list, we will add an instance variable that represents the last node of the linked list. We call this the **tail reference**.

This way we will have direct access to the last node, without having to traverse the list. We will call that instance variable representing the last node in the list *tail*.

When inserting a new *Player*, our *enqueue* method will perform the following operations:

1. Instantiate a new node containing the *Player* to be inserted.
2. Attach that new node at the end of the list, i.e., make the last node in the list, *tail*, point to that new node.
3. Mark the new node so that it is the last node of the list, i.e., assign that node to *tail*.
4. Increase the number of items by 1.

Figure 13.10a to **Figure 13.10d** illustrate the *enqueue* operation.

Figure 13.10a Enqueueing: Our Original Queue

Description

Figure 13.10b Enqueueing: Our Queue and the New Node

Description

Figure 13.10c Enqueueing: The New Node Has Been Attached at the End of the Queue

Description

Figure 13.10d Enqueueing: *tail* Has Been Updated

Description

Example 13.11 shows our *PlayerQueueLinkedList* class. This class also extends and inherits the functionality of our *ShellLinkedList* class.

Example 13.11 The *PlayerQueueLinkedList* Class

```java
/* The PlayerQueueLinkedList class
   Anderson, Franceschi
*/

public class PlayerQueueLinkedList extends ShellLinkedList
{
  // head and numberOfItems are inherited instance variables
  private PlayerNode tail; // last node

  public PlayerQueueLinkedList( )
  {
    super( );
    tail = null;
  }

   /** enqueue method
   * @param p Player object to insert
   */
   public void enqueue( Player p )
   {
      // insert as tail
      PlayerNode pn = new PlayerNode( p );
      if ( isEmpty( ) )
      {
        tail = pn;
        head = pn;
      }
      else
      {
        tail.setNext( pn );
        tail = pn;
      }
      numberOfItems++;
   }

   /** dequeue method
   *   @return    a reference to the Player object deleted
   */
   public Player dequeue( ) throws DataStructureException
   {
     if ( isEmpty( ) )
        throw new DataStructureException
          ( "empty queue: cannot dequeue" );
     else
     {
       Player deleted = head.getPlayer( );
       head = head.getNext( );
       if ( --numberOfItems == 0 )
           tail = null;
       return deleted;
     }
    }

    /** peek method
    *   @return    a reference to the Player object retrieved
    */
    public Player peek( ) throws DataStructureException
    {
      if ( isEmpty( ) )
        throw new DataStructureException
          ( "empty queue: cannot peek" );
      else
      {
        return head.getPlayer( );
      }
   }
}
```

Description

The constructor, from lines 10 to 14, calls the constructor of the superclass, and because it constructs an empty list, sets *tail* to *null*. The *peek* method is identical to the *peek* method of our *PlayerStackLinkedList* class, except for the message passed to the *DataStructureException* constructor.

The *enqueue* method, which we coded at lines 16–34, inserts an item at the end of the list. We first instantiate a *PlayerNode* object reference named *pn* at line 22, using the parameter *Player p* of the *enqueue* method.

Because we insert at the end of the list, we must properly handle the case when the queue is empty, in which case *tail* is *null*. We test if the queue is empty at line 23. If it is, we assign *pn* to *head* and *tail* at lines 25 and 26. After we execute these two lines, the queue contains one element, and that element is both the first and last item in the queue.

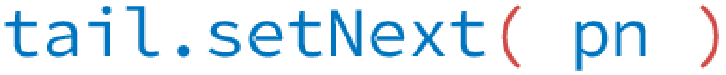
If the list is not empty, control skips to line 30, where we attach *pn* at the end of the list by setting the *next* instance variable of *tail* to *pn*. We then assign *pn* to *tail* in order to reflect that *pn* is now the last node of the list. Finally, and in all cases (empty list or not), we increment *numberOfItems* by 1 at line 33.

Figures 13.10a to 13.10d show the impact on the list of executing lines 22, 30, and 31 step by step.

COMMON ERROR TRAP

Before inserting or deleting an item in a linked list representing a queue, always check if the linked list is empty. Not doing so results in a *NullPointerException* at run time.

It is important to test if a queue is empty when coding the *enqueue* method. Indeed, if the queue is empty, then both *head* and *tail* are *null*. The code



```
tail.setNext( pn )
```

Description

at line 30 would, in this case, generate a *NullPointerException*.

The *dequeue* method, lines 36–52, is identical to the *pop* method of a linked list implementing a stack, except for lines 48 and 49. After decrementing *numberOfItems*, we check whether the list is empty. If so, we set *tail* to *null*.

A similar program to Example 13.9 can be coded to test all possible scenarios when using the methods of the *PlayerQueueLinkedList* class. This is proposed in the short program section of the exercises.

with these end-of-chapter questions

13.14.1	Multiple Choice Exercises Questions 2, 3, 4, 5, 6, 7
13.14.6	Write a Short Program Questions 52, 53
13.14.8	Technical Writing Question 73

	index	***Player* object**
top	2	(8, Gino, Diablo)
	1	(7, Sarah, Mario)
	0	(2, Jin, Golf)

	index	***Player* object**
top	3	(6, Steve, NFL)
	2	(8, Gino, Diablo)
	1	(7, Sarah, Mario)
	0	(2, Jin, Golf)

	index	*Player* object
	3	(6, Steve, NFL)
top	2	(8, Gino, Diablo)
	1	(7, Sarah, Mario)
	0	(2, Jin, Golf)

13.5 Array Representation of Stacks

Earlier in this chapter, we discussed how a stack can be represented by a linked list. Since a stack is a last in, first out data structure, we coded the *push* (insert) and *pop* (delete) methods of the linked list to insert or delete at the beginning of the list. Linked lists offer the advantage of being expandable one object at a time, so we do not have to worry about running out of capacity.

However, if we know in advance that the number of objects on a stack will always be less than some maximum number, we can represent the stack using an array, which is easier to implement.

Table 13.6 shows the APIs of the *push*, *pop*, and *peek* methods for a stack implemented using an array.

ArrayStack Methods

Methods of the *ArrayStack* Class	
Return value	**Method name and argument list**

`boolean`	`push( Player p )` inserts *Player p* at the top of the stack, if the stack is not full. Returns *true* if the insertion was successful; *false* otherwise.
`Player`	`pop( )` removes and returns the *Player* at the top of the stack, if the stack is not empty. If the stack is empty, the method throws a *DataStructureException*.
`Player`	`peek( )` returns the *Player* at the top of the stack if the stack is not empty. If the stack is empty, the method throws a *DataStructureException*.

Description

To match the LIFO functionality of a stack, we instantiate the array with the maximum number of elements. We add items to the stack starting at index 0, storing the items in adjacent locations in the array. To keep track of the array index of the last element inserted, we maintain an index **top**, short for “top of the stack.” We always remove (*pop*) the item at the top of the stack.

To push an item onto the stack, we increment the value of *top* by 1 and store the element at the new *top* index. To pop an item from the stack, we return the item at index *top* and decrement the value of *top* by 1.

Figure 13.11a shows how we can visualize a stack of *Players*. **Figure 13.11b** and **13.11c** show the stack after pushing a *Player* (*6*, *Steve*, *NFL*) and then popping one element. Figure 13.11c shows that the array element at index 3 is still *Player* (*6*, *Steve*, *NFL*), but that is irrelevant. Since *top* has the value 2, the element at index 3 is not on the stack. When the next item is pushed onto the stack, we will reuse that element.

Figure 13.11a Our Original Stack

Description

Figure 13.11b Our Stack After Pushing Player (6, Steve, NFL)

Description

Figure 13.11c Our Stack After Popping Once

Description

One disadvantage of implementing a stack with an array is that the array has a fixed size, and it is possible that the array can be filled completely with elements of the stack. Thus, our *push* method needs to test if the array is full before pushing an element onto the stack. Similarly, our *pop* method needs to test if the array is empty before popping an element from the stack.

Example 13.12 shows our *ArrayStack* class.

Example 13.12 The *ArrayStack* Class

```
/* The ArrayStack class
   Anderson, Franceschi
*/

public class ArrayStack
{
  private static final int STACK_SIZE = 100; // maximum array size
  private Player [ ] stack;            // array of Player objects
  private int top;             // last used index; top of the stack

  public ArrayStack( )
  {
    stack = new Player[STACK_SIZE];
    top = -1; // stack is empty
  }

   /** push method
   * @param p Player object to insert
   * @return    true if insertion was successful false otherwise
   */
   public boolean push( Player p )
   {
     if ( !isFull( ) ) // is there room to insert?
     {
       stack[++top] = p;
       return true;
     }
     else
       return false;
   }

   /** pop method
   *   @return    the Player deleted
   */
   public Player pop( ) throws DataStructureException
   {
     if ( !isEmpty( ) ) // is there an item to delete?
       return stack[top--];
     else
       throw new DataStructureException
                   ( "Stack empty: cannot pop" );
   }

   /** peek method
   *   @return    the Player at the top of the stack
   */
   public Player peek( ) throws DataStructureException
   {
     if ( !isEmpty( ) ) // stack is not empty
       return stack[top];
     else
       throw new DataStructureException
                   ( "Stack empty: cannot peek");
   }

   /** isEmpty method
   *   @return    true if stack is empty, false otherwise
   */
   public boolean isEmpty( )
   {
     return ( top == -1 );
   }

   /** isFull method
   *   @return    true if stack is full, false otherwise
   */
   public boolean isFull( )
   {
      return ( top == ( STACK_SIZE - 1 ) );
   }

  /** toString method
  *   @return the stack elements starting at top
  */
  @Override
  public String toString( )
  {
    String stackString = "";
    for ( int i = top; i >= 0; i-- )
      stackString += ( i + ": " + stack[i] + "\n" );
    return stackString;
  }
}
```

Description

We declare *STACK_SIZE*, *stack*, and *top*, our three fields at lines 7–9.

Stack is an array of *Players*. *STACK_SIZE* is the size of the array *stack*. *Top* represents the index of the element of the array *stack* that is at the top of the stack. The value of *top* will vary from *–1* (when the stack is empty) to *STACK_SIZE – 1* (when the stack is full).

In the default constructor, coded at lines 11–15, we instantiate *stack* and then set *top* to –1, which indicates that the stack is empty. When a client program pushes the first *Player* onto the stack, *top* will be incremented, so that the top of the stack will be the array element at index 0.

We coded the *push* method at lines 17–30. The *push* method returns *true* (line 26) if the stack is not full before we insert, and *false* (line 29) if it is, in which case we cannot insert. We test if the stack is not full at line 23. If it is not full, we use the prefix - auto-increment operator to combine two operations at line 25: first increment *top* by 1, then assign *p*, the *Player* parameter of the *push* method, to the element at index *top*.

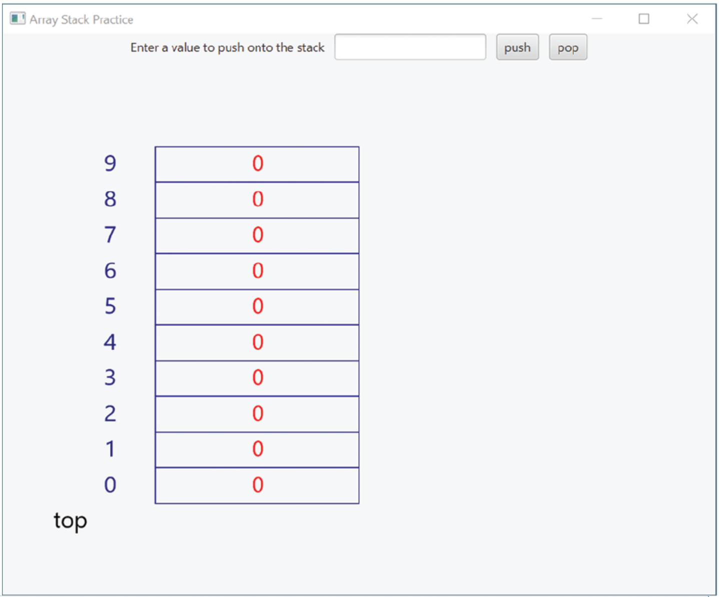
We coded the *pop* method at lines 32–42. The *pop* method attempts to delete and return a *Player* object from the top of the stack. The method *throws* a *DataStructureException* at lines 40–41 if the stack is empty, in which case we cannot pop. If it is not empty, we use the postfix auto-decrement operator to combine two operations at line 38: first return the *Player* stored at index *top* in the array *stack*, then decrement *top* by 1.

COMMON ERROR TRAP

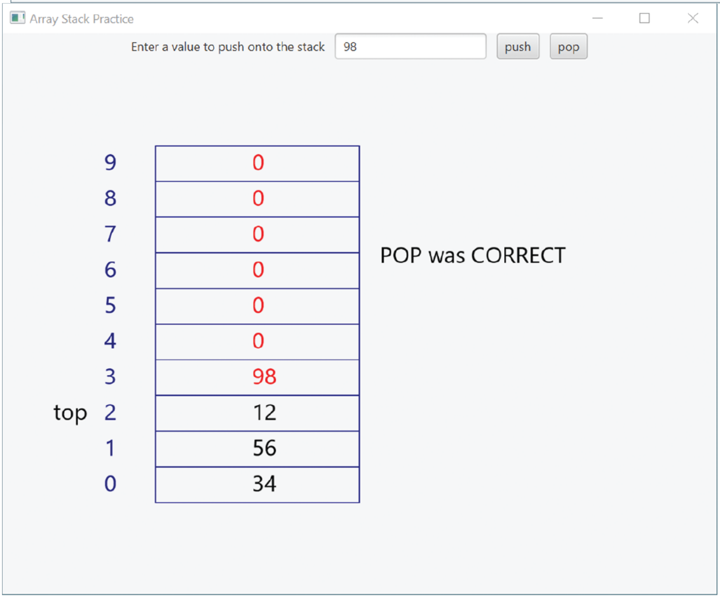
Do not confuse the top of the stack with the last index in the array. Array elements with an index higher than *top* are not on the stack.

We have also coded a few other methods in this class. The *peek* method (lines 44–54) is similar to *pop*, except that it does not delete from the stack and it returns the element at the top of the stack. Again, this enables the client to directly update that object in the stack. The *isEmpty* and *isFull* methods are coded at lines 56–62 and 64–70, respectively. And the *toString* method, coded at lines 72–82, returns a *String* representation of the contents of the stack. Note that in that method, we loop from *top* to 0, not from *STACK_SIZE – 1* to 0.

As before, a program similar to Example 13.9 can be coded to test all possible scenarios on the methods of the *ArrayStack* class. This is proposed in the short program section of the exercises.


Array Stack Practice
Enter a value to push onto the stack
push
pop

9	0
8	0
7	0
6	0
5	0
4	0
3	0
2	0
1	0
0	0
top	


Array Stack Practice
Enter a value to push onto the stack
push
pop

98

	9	0
	8	0
	7	0
	6	0
	5	0
	4	0
	3	98
top	2	12
	1	56
	0	34

POP was CORRECT

13.6 Programming Activity 1: Writing Methods for a Stack Class

In this activity, you will work with a stack represented by an array, performing this activity:

> Code the *push* and *pop* methods to insert onto and delete from a stack represented by an array of *ints*.

The framework will animate your code to give you feedback on the correctness of your code. It will display the state of the stack at all times. The result of your operation will be displayed, reflecting the value returned by your *push* or *pop* method. The items in the stack will be displayed in black while the array elements that are not part of the stack will be displayed in red.

Instructions

Copy the contents of the Programming Activity 1 folder for this chapter from the companion website for this text onto a folder on your computer. Open the *StackArray.java* source file. Searching for five asterisks (*****) in the source code will position you to the code section where you will add your code.

In this task, you will fill in the code inside the methods *push* and *pop* to insert onto and delete from a stack. **Example 13.13** shows the section of the *StackArray* source code where you will add your code. This example is different from the one in the chapter. The stack is an array of *ints*, not *Players*. The *isFull* and *isEmpty* methods have not been provided; you can code them or not, depending on how you want to implement the *push* and *pop* methods.

Example 13.13 Location of Student Code in *StackArray* Class

```
/** push method
*   @param    value   value to be pushed onto the stack
*   @return   true if successful, false if unsuccessful
*/
public boolean push( int value )
{
 // ***** 1. Student code starts here *****
 // stack is an int array instance variable representing
 // the array that stores our stack

 // top is an instance variable representing
 // the index of the top of the stack

 // CAPACITY is a static constant representing
 // the size of the array stack

 // The push method adds the argument value
 // to the top of the stack, if it is possible
 // code the push method here
 // end of student code, part 1
}

/** pop method
*   @return   the value of the top element of the stack, if
*             successful
*/
public int pop( ) throws DataStructureException
{
 // ***** 2. Student code restarts here *****
 // stack is an int array instance variable representing
 // the array that stores our stack

 // top is an instance variable representing
 // the index of the top of the stack

 // CAPACITY is a static constant representing
 // the size of the array stack

 // The pop method deletes the element
 // at the top of the stack, if it is possible

// code the pop method here

  // end of student code, part 2
}
```

Description

To test your code, compile *StackArray.java* and run the *StackPracticeApplication* class, which contains the *main* method. When the program begins, a window will display the state of the stack, along with two buttons labeled “push” and “pop,” as shown in **Figure 13.12**.

Figure 13.12 Opening Window

Description

Enter a value into the text box, then click on the “push” button to insert onto the stack. Click on the “pop” button to delete from the stack. Close the window to exit the program.

If you successively push 34, 56, 12, and 98 onto the stack, then pop once, the window will look like the one shown in **Figure 13.13**.

Figure 13.13 Sample Window After Performing Some Stack Operations

Description

Troubleshooting

If your method implementation does not animate or animates incorrectly, check these items:

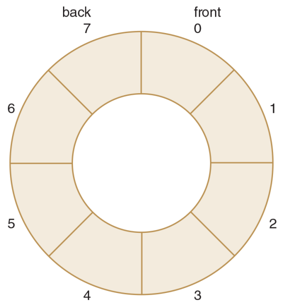
- Check the feedback in the window to see if your code gives the correct result.
- Verify that you updated the value of *top* correctly.
- Verify that you correctly coded the cases where the stack is full (*push* method) and the stack is empty (*pop* method).

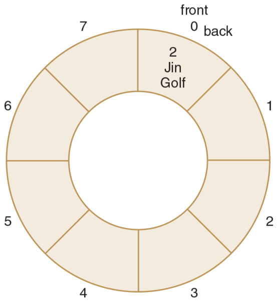
DISCUSSION QUESTIONS

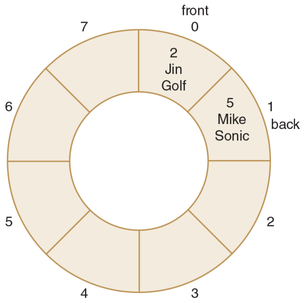
1. Explain how the array elements above the index *top* can have assigned values, but are still irrelevant.
2. Explain what happens if you do not test whether the stack is empty in the *pop* method or full in the *push* method.

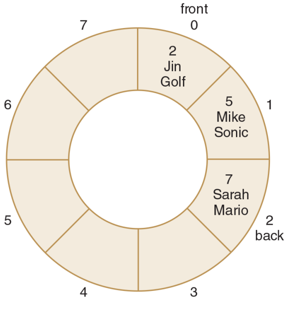
	index	***Player* object**
	7	
	6	
	5	
back	4	(6, Steve, NFL)
	3	(8, Gino, Diablo)
	2	(7, Sarah, Mario)
	1	(2, Jin, Golf)
front	0	(5, Ajay, Sonic)

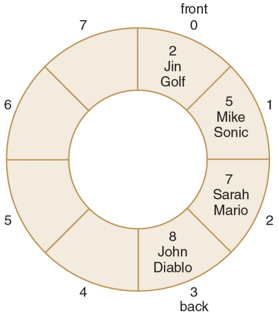
	index	***Player* object**
	7	
	6	
	5	
back	4	(6, Steve, NFL)
	3	(8, Gino, Diablo)
	2	(7, Sarah, Mario)
front	1	(2, Jin, Golf)
	0	(5, Ajay, Sonic)

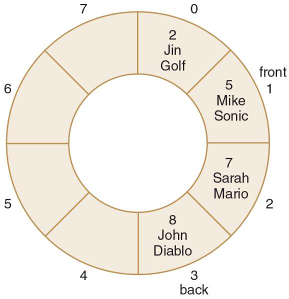

back
front

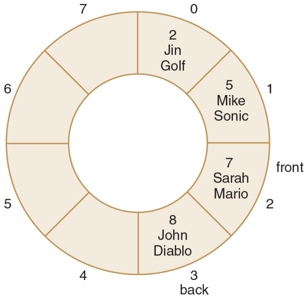

front
0 back
Jin
Golf

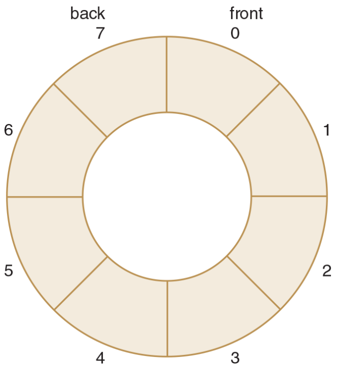

front
Jin
Golf
Mike
Sonic
back

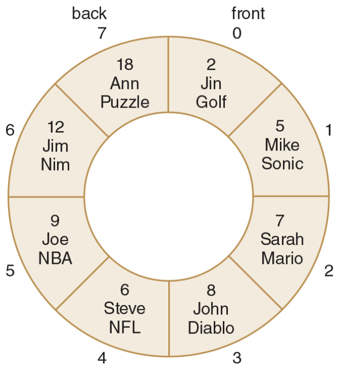

front
0
7
2
Jin
Golf
5
Mike
Sonic
1
7
Sarah
Mario
2
back
3
4
5
6


front
0
7
2
Jin
Golf
5
Mike
Sonic
1
7
Sarah
Mario
2
8
John
Diablo
3
back
4
5
6


0
7
2
Jin
Golf
front
1
5
Mike
Sonic
7
Sarah
Mario
2
8
John
Diablo
3
back
4
5
6


0
7
2
Jin
Golf
1
5
Mike
Sonic
front
7
Sarah
Mario
2
8
John
Diablo
3
back
4
5
6


back
front
7
0
1
2
3
4
5
6


back
front
7
0
18 Ann Puzzle
2 Jin Golf
12 Jim Nim
5 Mike Sonic
9 Joe NBA
7 Sarah Mario
6 Steve NFL
8 John Diablo
1
2
3
4
5
6

13.7 Array Representation of Queues

Earlier in this chapter, we also saw how a queue can be represented by a linked list. Again, if we know in advance that the number of objects in a queue will always be less than some maximum number, we can also use an array to represent the queue.

To match the FIFO functionality of a queue, we will need to keep track of two things:

1. the location of the back of the queue. This is the index of the last element enqueued or added to the queue. We will call the index of that element *back*.
2. the location of the front of the queue. That is the index of the element that will be dequeued or retrieved next. We will call the index of that element *front*.

The queue will be comprised of the elements whose indexes are between *back* and *front*, inclusive.

To dequeue, or delete from the queue, we will return the item at index *front* and increase the value of *front* by one. To enqueue, or insert an element in the queue, we will increment the value of *back* by one, and insert the element at the array index *back*.

There is one important problem in representing a queue with a standard array: the number of available elements for the queue in the array will shrink over time as we enqueue and dequeue, since enqueueing and dequeueing both advance their indexes toward the end of the array.

To illustrate this point, let's consider a queue represented by an array of eight elements. We start by enqueueing five players in this order: (*5*, *Ajay*, *Sonic*), (*2*, *Jin*, *Golf*), (*7*, *Sarah*, *Mario*), (*8*, *Gino*, *Diablo*), and (*6*, *Steve*, *NFL*). Since (*5*, *Ajay*, *Sonic*) was the first to be inserted in the queue, that *Player* is now at the front of the queue. (*6*, *Steve*, *NFL*), inserted last, is at the back of the queue. Thus, (*5*, *Ajay*, *Sonic*) will be stored at index 0 and (*6*, *Steve*, *NFL*) will be stored at index 4, as shown in **Figure 13.14a**. Suppose now that we dequeue once. *Front* now has the value 1, as shown in **Figure 13.14b**. The array element at index 0 is no longer in the queue and its value is irrelevant. Since we insert at the back, the array element at index 0 can no longer be used for the queue. If we dequeue again, *front* will have the value 2, and we will no longer be able to use the array element at index 1. As we keep enqueueing and dequeueing, the values of *back* and *front* keep increasing and we have less and less usable space in the array. Indeed, when *back* reaches 7, we will no longer be able to enqueue at all.

Figure 13.14a Our Queue After Enqueueing the First Five Elements

Description

Figure 13.14b Our Queue After Dequeueing Once

Description

SOFTWARE ENGINEERING TIP

When implementing a queue as an array, think of it as a circular array.

There is a solution to this problem: deal with the array as if it were circular. After *back* reaches the last index of the array, we start enqueueing again at index 0. Thus, in a circular array, the next index after the last array index is 0. Let's say that at one point the *back* marker reaches 7 and the *front* marker is at 5. When we enqueue a new object, we will store that object at index 0, which is the "next" index after 7 if we imagine that the array is circular. In this way, our useful array capacity never shrinks and is always 8.

How do we know that we have reached the last array index and that the next index should be 0? We simply add 1 to the value of *back*, and then take that number modulo the size of the array, which we call QUEUE_SIZE.

Table 13.7 shows the APIs of the *enqueue* and *dequeue* methods.

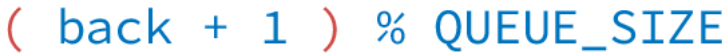
ArrayQueue Methods

Methods of the *ArrayQueue* Class

Return value	**Method name and argument list**
`boolean`	`enqueue( Player p )` inserts *Player p* at the back of the queue if the queue is not full. Returns *true* if the insertion was successful, *false* otherwise.
`Player`	`dequeue( )` returns and removes the *Player* at the front of the queue. If the queue is empty, the method throws a *DataStructureException*.
`Player`	`peek( )` returns the *Player* at the front of the queue. If the queue is empty, the method throws a *DataStructureException*.

Description

Figure 13.15 illustrates a sequence of insertions and deletions in a queue of *Players* implemented as a circular array. When we begin, the queue is empty. The value of *front* is 0 and the value of *back* is *QUEUE_SIZE – 1*. When we enqueue the first item, that element is placed at index



```
( back + 1 ) % QUEUE_SIZE
```

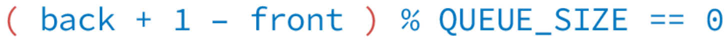
Description

which is now 0, and *back* will be given the value 0. If we enqueue again, the new element will be placed at index 1 and *back* will be given the value 1. If we enqueue two more items, they will be placed at indexes 2 and 3, respectively, and back will be given the value 3. If we then dequeue, we will return the item at index 0, and *front* will become 1. If we dequeue again, we will return the item at index 1, and *front* will become 2.

Figure 13.15 Starting with an Empty Queue, Four Successive Enqueues Followed by Two Dequeues

Description

When we enqueue, we first need to check if the queue is full. When the queue is full, the relationship between *back* and *front* is:

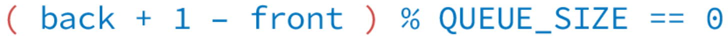


```
( back + 1 - front ) % QUEUE_SIZE == 0
```

Description

For example, in a full queue with 8 elements, the values of *front* and *back* could be 0 and 7, respectively, or they could be 5 and 4 or any other pair of values for which the expression above is *true*.

When we dequeue, we first need to check if the queue is empty. When the queue is empty, the relationship between *front* and *back* is the same as when the queue is full:



```
( back + 1 - front ) % QUEUE_SIZE == 0
```

Description

Indeed, when there is only one item in the queue, *back* and *front* have the same index value. When we dequeue that last item from the queue, *front* will increase by 1 modulo *QUEUE_SIZE*, resulting in the preceding relationship between *front* and *back*. **Figure 13.16** shows an example of an empty queue and a full queue.

Figure 13.16 An Empty Queue and a Full Queue

Description

So, how do we know if the queue is full or empty? In order to distinguish a full queue from an empty queue, we must add another instance variable to our class. We will keep track of the number of elements in the queue: if the number of elements is 0, then the queue is empty; if the number of elements is equal to the size of the array, then the queue is full.

COMMON ERROR TRAP

Do not confuse array indexes 0 and *QUEUE_SIZE – 1* with *front* and *back*. In a queue represented by a circular array, the indexes 0 and *QUEUE_SIZE – 1* are irrelevant.

Example 13.14 shows our *ArrayQueue* class.

Example 13.14 The *ArrayQueue* Class

```java
/* The ArrayQueue class
   Anderson, Franceschi
*/

public class ArrayQueue
{
   private static final int QUEUE_SIZE = 8;
   private Player [ ] queue;
   private int front;
   private int back;
   private int numberOfItems;

   public ArrayQueue( )
   {
     queue = new Player[ QUEUE_SIZE ];
     front = 0;
     back = QUEUE_SIZE - 1;
     numberOfItems = 0;
   }

   public boolean isFull( )
   {
     return ( numberOfItems == QUEUE_SIZE );
   }

   public boolean isEmpty( )
   {
     return ( numberOfItems == 0 );
   }

   /** enqueue method
   *   @param    p    the Player to insert
   *   @return   true if list is not full, false otherwise
   */
   public boolean enqueue( Player p )
   {
     if ( !isFull( ) )
     {
       queue[( back + 1 ) % QUEUE_SIZE] = p;
       back = ( back + 1 ) % QUEUE_SIZE;
       numberOfItems++;
       return true;
     }
     else
       return false;
   }

   /** dequeue method
   *   @return    the Player deleted
   */
   public Player dequeue( ) throws DataStructureException
   {
     if ( !isEmpty( ) )
     {
        front = ( front + 1 ) % QUEUE_SIZE;
        numberOfItems--;
        return queue[( QUEUE_SIZE + front - 1 ) % QUEUE_SIZE];
     }
     else
       throw new DataStructureException
                   ( "Queue empty: cannot dequeue" );
   }

   /** toString method
   * @return a front-to-back String representation of the queue
   */
   @Override
   public String toString( )
   {
      String queueString = "";
      for ( int i = front; i < front + numberOfItems; i++ )
       queueString += queue[i % QUEUE_SIZE].toString( ) + "\n";
      return queueString;
   }
}
```

Description

In the constructor, coded at lines 13–19, we instantiate the array *queue*, set *front* to 0, *back* to *QUEUE_SIZE − 1*, and *numberOfItems* to 0. When the first element is inserted in the queue, *back* will be increased by 1 modulo *QUEUE_SIZE* and its value will become 0.

The *isFull* and *isEmpty* methods, coded at lines 21–24 and 26–29, enable a client program to check if the queue is full or empty before enqueueing or dequeueing a *Player*. Our *enqueue*, *dequeue*, and *toString* methods also call these methods.

In the *enqueue* method, coded at lines 31–46, we attempt to insert a *Player* into the queue. The *enqueue* method returns *false* if the queue is full (line 45) to indicate that we cannot insert. If the queue is not full, we place the *Player* at the back of the queue, update *back* accordingly, increment the number of items, and return *true* (lines 39–42).

In the *dequeue* method, coded at lines 48–62, we attempt to delete and return a *Player* from the front of the queue. The method *throws* a *Data StructureException* at lines 60–61 if the queue is empty, in which case there are no *Players* to delete. If the queue is not empty, we update *front*, decrement the number of items, and return the *Player* that was at the front of the queue (lines 55–57). Note that we add *QUEUE_SIZE* to the expression *front + 1* to guarantee that it will be nonnegative.

We could also code a *peek* method. It would be similar to the *peek* method we coded for the *StackArray* class, except that *top* would be replaced by *front*. Coding the *peek* method is included as an exercise at the end of the chapter.

The *toString* method, coded at lines 64–74, is slightly different from the *toString* methods we have written so far. Since we know that there are *numberOfItems* items in the queue and that the first item is at index *front*, we can simply start at *front* and loop *numberOfItems* times to build our *String* representation of the queue. Depending on how many items are in the queue and the value of *front*, the looping variable could get larger than *QUEUE_SIZE − 1*, so we use the modulus operator (line 72) to make sure we have a valid index.

As before, a very similar program to Example 13.9 can be coded to test all possible scenarios on the methods of the *ArrayQueue* class. This is proposed in the short program section of the exercises.

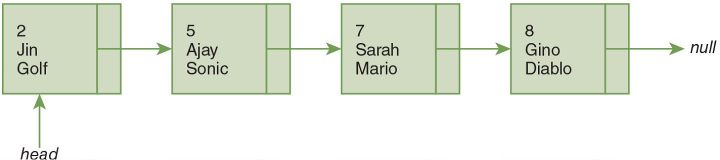
As we have demonstrated, a stack or queue can be implemented using either an array or a linked list. Each implementation has advantages and disadvantages. Arrays are easier to code and every item in the stack or queue can be accessed directly through its index. Linked lists are easily expanded one item at a time. To expand an array, we would need to instantiate a new, larger array and copy the elements of the existing stack or queue to the new array, which is quite tedious.

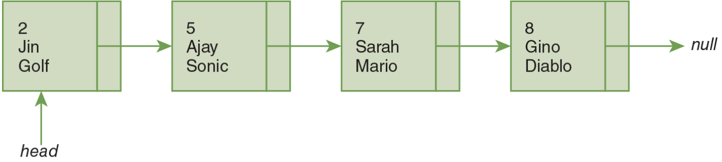
Table 13.8 summarizes these trade-offs.

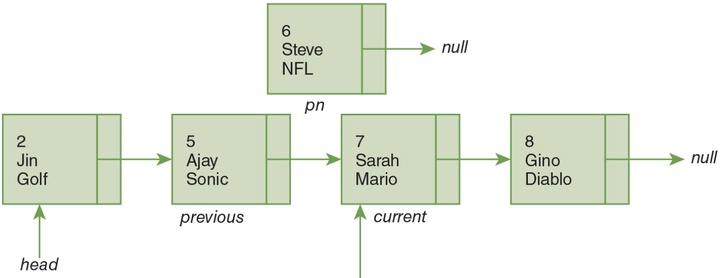
Array Versus Linked-List Implementation of a Stack or a Queue

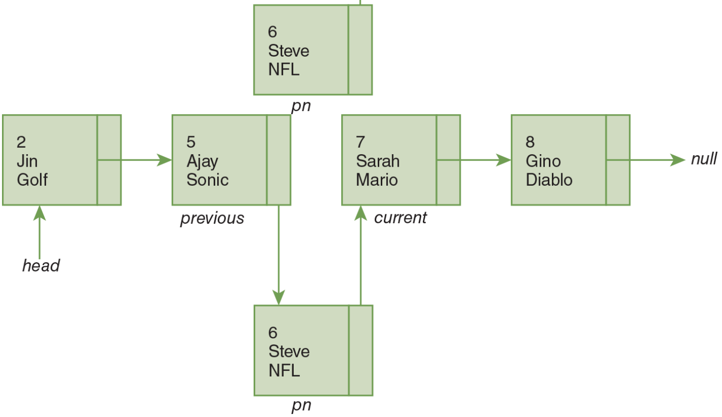
	Array	**Linked List**
Easily expanded	No	Yes
Direct access to every item	Yes	No
Easy to code	Yes	No

Description


2
Jin
Golf
5
Ajay
Sonic
7
Sarah
Mario
8
Gino
Diablo
null
head


2
Jin
Golf
5
Ajay
Sonic
7
Sarah
Mario
8
Gino
Diablo
null
head


6
Steve
NFL
null
pn
2
Jin
Golf
5
Ajay
Sonic
7
Sarah
Mario
8
Gino
Diablo
null
previous
current
head


6
Steve
NFL
pn
2
Jin
Golf
5
Ajay
Sonic
7
Sarah
Mario
8
Gino
Diablo
null
previous
current
head
6
Steve
NFL
pn

```
Number of items in the list: 0

inserting id: 7        name: Sarah        game: Mario
Number of items in the list: 1
id: 7   name: Sarah     game: Mario

inserting id: 2        name: Jin          game: Golf
Number of items in the list: 2
id: 2   name: Jin       game: Golf
id: 7   name: Sarah     game: Mario

inserting id: 5          name: Ajay        game: Sonic
Number of items in the list: 3
id: 2   name: Jin       game: Golf
id: 5   name: Ajay      game: Sonic
id: 7   name: Sarah     game: Mario

inserting id: 8          name: Gino        game: Diablo
Number of items in the list: 4
id: 2   name: Jin       game: Golf
id: 5   name: Ajay      game: Sonic
id: 7   name: Sarah     game: Mario
id: 8   name: Gino      game: Diablo
```

13.8 Sorted Linked Lists

Let’s go back to our linked list of video game players. If we want to display that list on a website so that all the players can see it, we might want to display the list in ascending (or descending) order by *id* number, or in alphabetical order by name or game. If we store the items in the list in sorted order, we can display the list by simply calling the *toString* method.

Often, a class includes a method that enables us to compare two objects of that class. That method is often named *compareTo*. It returns a positive number if *this* object is greater than the parameter object, *0* if they are equal, and a negative number if *this* object is less than the parameter object. The *Comparable* interface defines such a method. It is standard practice for a class to implement *Comparable* and override its *compareTo* method. Because the *Comparable* interface uses generics, the *Player* class header becomes:

```
public class Player implements Comparable<Player>
```

Description

The *compareTo* method is added to our *Player* class as shown in **Example 13.15**.

Example 13.15 The *compareTo* Method Is Added to the *Player* Class

```
...
105
106  /** compareTo method
107  *  @param p a Player to compare to this Player
108  *  @return  an int, -1, 0, or 1 as this Player is less than,
109  *                   equal to, or greater than player
110  */
111  public int compareTo( Player p )
112  {
113     if ( id == p.id && name.equals( p.name )
114                     && game.equals( p.game ) )
115        return 0;
116     else if ( id > p.id ||
117               ( id == p.id && name.compareTo( p.name ) > 0  ) ||
118               ( id == p.id && name.equals( p.name )
119                      && game.compareTo( p.game ) > 0  )  )
120        return 1;
121     else
122        return -1;
123  }
124 }
```

Description

A *Player* is greater than another *Player* if (lines 116–119)

- its *id* is greater, or
- its *id* is the same but its name is greater (in alphabetical order), or
- its *id* and name are the same and its game is greater than the one of the other *Player*.

A linked list that stores its items in ascending order (or descending order) according to a criterion is called a **sorted linked list**. Without loss of generality, we will consider a linked list sorted in ascending order.

Table 13.9 shows the APIs of the *insert* and *delete* methods for a sorted linked list. The only difference in this API from that of our unsorted list is that the location for inserting an element is dependent on the key value, rather than always inserting at the beginning of the list.

PlayerSortedLinkedList Methods

Methods of the *PlayerSortedLinkedList* Class

Return value	**Method name and argument list**
`void`	`insert( Player p )` inserts *Player p* in a location that keeps the list sorted in ascending order.
`Player`	`delete( int searchID )` returns and removes the first *Player* of the list with an *id* equal to *searchID*. If there is no such *Player* on the list, the method throws a *DataStructureException*.

Description

By default, an empty list is sorted, so a newly instantiated list is sorted. As we add elements, we need to maintain the sorted order of the list. Thus, the *insert* method must locate the proper position for inserting each element so that the inserted element's *id* is greater than the *id* of the previous element (if any) and less than or equal to the *id* of the next element (if any). We will find that proper place by traversing the list, comparing the value of the *id* of the new *Player* with the values of the *ids* of the *Players* stored at the various nodes in the list.

If the value of the item to insert will place it at the beginning of the list, then we will insert it in the same manner as we did in our earlier examples.

When inserting a new *Player* in the middle or at the end of the list, our *insert* method will do the following:

1. Instantiate a new node containing the *Player* to be inserted.
2. Traverse the list to identify the location to insert the new node. We will call the node before the insertion point *previous*, and the node after the insertion point *current*.
3. Attach the new node to *current*; that is, make the new node point to *current*.
4. Attach *previous* to the new node; that is, make *previous* point to the new node.
5. Increase the number of items in the list by 1.

Figures 13.17a to **13.17d** illustrate inserting a node somewhere in the middle of the sorted list.

Figure 13.17a Our Original Sorted List Before Inserting *Player (6, Steve, NFL*)

Description

Figure 13.17b Step 1: Instantiate the New Node

Description

Figure 13.17c Steps 2 and 3: Insert Occurs Between *previous* and *current*. Attach the New Node to *current*

Description

Figure 13.17d Step 4: Attach *previous* to the New Node

Description

The insertion code corresponding to Figures 13.17c and 13.17d is shown in **Example 13.16** at lines 38–39.

Keeping the list in sorted order also impacts our *delete* method. If the item we are looking for is not in the list, we may be able to determine that fact without traversing the entire list. As soon as we visit an item with a value greater than the key value, we know that the item we are looking for is not in the list. Because the list is sorted in ascending order, all the *Players* stored after that node must have an *id* value greater than the key. Thus, we will be able to exit our *delete* method at this point, saving processing time.

Example 13.16 shows our *PlayerSortedLinkedList* class. This class also extends and inherits the functionality of our *ShellLinkedList* class. The list is sorted in ascending order according to the value of each *Player's id*.

Example 13.16 The *PlayerSortedLinkedList* Class

```
 /* The PlayerSortedLinkedList class
    Anderson, Franceschi
*/

public class PlayerSortedLinkedList extends ShellLinkedList
{
  // head and numberOfItems are inherited instance variables

  public PlayerSortedLinkedList( )
  {
    super( );
  }

  /** insert method
  *   @param    p    Player object to insert
  */
  public void insert( Player p )
  {
    PlayerNode pn = new PlayerNode( p );

    // we will insert after previous and before current
    PlayerNode current = head;
    PlayerNode previous = null;
    while ( current != null
          && current.getPlayer( ).compareTo( p ) < 0 )
    {
      previous = current;
      current = current.getNext( );
    }

    if ( previous == null ) // insert as head
    {
      pn.setNext ( head );
      head = pn;
    }
    else
    {
      pn.setNext ( current );
      previous.setNext ( pn );
    }
    numberOfItems++;
  }

  /** delete method
  *   @param    searchID    id of Player to delete
  *   @return the Player deleted
  */
  public Player delete( int searchID )
                       throws DataStructureException
  {
    PlayerNode current = head;
    PlayerNode previous = null;
    while ( current != null
            && current.getPlayer( ).getID( ) != searchID )
    {
      if ( current.getPlayer( ).getID( ) > searchID )
           throw new DataStructureException
               ( searchID + " not found: cannot be deleted" );
      previous = current;
      current = current.getNext( );
    }

    if ( current == null ) // not found
           throw new DataStructureException
              ( searchID + " not found: cannot be deleted" );
    else       // searchID found at Player at node current
    {
      if ( current == head )
           head = head.getNext( ); // delete head
      else
           previous.setNext ( current.getNext ( ) );

      numberOfItems--;
      return current.getPlayer( );
     }
   }
 }
```

Description

The *insert* method, which we coded at lines 14 to 42, inserts a node containing its *Player* parameter *p*. Line 19 declares and instantiates a *PlayerNode* object, called *pn*, which we will insert in the linked list. To get ready to search for the insertion point for the new *Player*, we declare two *PlayerNode* object references, *current* and *previous*, at lines 22–23, and assign them *head* and *null*. We use *current* to traverse the list, going just past the point of insertion, and we use *previous* to track the node just before *current*. We will insert *pn* between *previous* and *current*. From lines 24 to 29, we use a *while* loop to traverse the list. We construct our *while* loop condition so that we will exit the loop if the list is empty or if we have reached the end of the list (we test if current is *null* at line 24), or if we are visiting a node containing a *Player* that is larger than or equal to *p*, the *Player* parameter of the *insert* method (line 25).

As mentioned earlier, there are two different cases for insertion: either we insert at the beginning of the list, or we insert in the middle or at the end of the list. At line 31 we test if *previous* is *null*, in which case we never entered the *while* loop because the list is empty or because the head node contains a *Player* whose *id* value is greater than *p's id*. Either way, we insert at the beginning of the list at lines 33 and 34.

If *previous* is not *null*, we will insert in the middle of the list or at the end of the list. To insert the node *pn* between *previous* and *current*, we connect *pn* to *current* at line 38, and *previous* to *pn* at line 39. Figures 13.17a to 13.17d show the step-by-step impact of lines 19, 38, and 39 on the sorted linked list.

The *delete* method (lines 44–76) is very similar to the *delete* method of the *PlayerLinkedList* class. The only difference is at lines 56–58. We first test at line 56 if the *id* of the *Player* at *current* is greater than *searchID*. If that is *true*, we have no chance of finding a *Player* object with an *id* of *searchID* since the list is sorted in ascending order. Therefore, we *throw* a *DataStructureException* with an appropriate message, and we exit the method.

Let's test our *PlayerSortedLinkedList* class. In order to keep things simple, we will test the *insert* method only, because the *delete* method is, as discussed, almost identical to the *delete* method of the *PlayerLinkedList* class.

We want to test the following cases:

- insert in an empty list
- insert at the beginning of the list
- insert in the middle of the list
- insert at the end of the list

We traverse the list after each insertion to check that the *Player* was inserted at the correct location in the sorted linked list.

Example 13.17 shows how to use the *PlayerSortedLinkedListTest* class and how to test its methods.

Example 13.17 The *PlayerSortedLinkedListTest* Class

```
/* The PlayerSortedLinkedListTest class
   Anderson, Franceschi
*/

public class PlayerSortedLinkedListTest
{
  public static void main( String [ ] args )
  {
    Player p1 = new Player( 7, "Sarah","Mario" );
    Player p2 = new Player( 2, "Jin","Golf" );
    Player p3 = new Player( 5, "Ajay","Sonic" );
    Player p4 = new Player( 8, "Gino","Diablo" );

    // construct empty PlayerSortedLinkedList
    PlayerSortedLinkedList players =
                  new PlayerSortedLinkedList( );

    System.out.println( "Number of items in the list: "
       + players.getNumberOfItems( ) + "\n" + players.toString( ) );

    System.out.println( "inserting " + p1 );
    players.insert( p1 ); // insert in empty list
    System.out.println( "Number of items in the list: "
       + players.getNumberOfItems( ) + "\n" + players.toString( ) );

    System.out.println( "inserting " + p2 );
    players.insert( p2 );    // insert at the beginning of the list
    System.out.println( "Number of items in the list: "
       + players.getNumberOfItems( ) + "\n" + players.toString( ) );

    System.out.println( "inserting " + p3 );
    players.insert( p3 ); // insert in the middle of the list
    System.out.println( "Number of items in the list: "
       + players.getNumberOfItems( ) + "\n" + players.toString( ) );

    System.out.println( "inserting " + p4 );
    players.insert( p4 ); // insert at the end of the list
    System.out.println( "Number of items in the list: "
       + players.getNumberOfItems( ) + "\n" + players.toString( ) );
  }
}
```

Description

In Example 13.17, we instantiate our usual four *Player* objects *p1*, *p2*, *p3*, and *p4* at lines 9–12. We choose the *id* values so that our four test cases will be covered when we successively insert the *Player* objects. We instantiate the *PlayerSortedLinkedList players* object at lines 14–16.

We first traverse the empty list at lines 18–19. Then, we successively insert *p1*, *p2*, *p3*, and *p4*, traversing the list after each insertion (lines 21–39). **Figure 13.18** shows the output of Example 13.17. As we can see, *players* remains sorted in ascending order after each insertion.

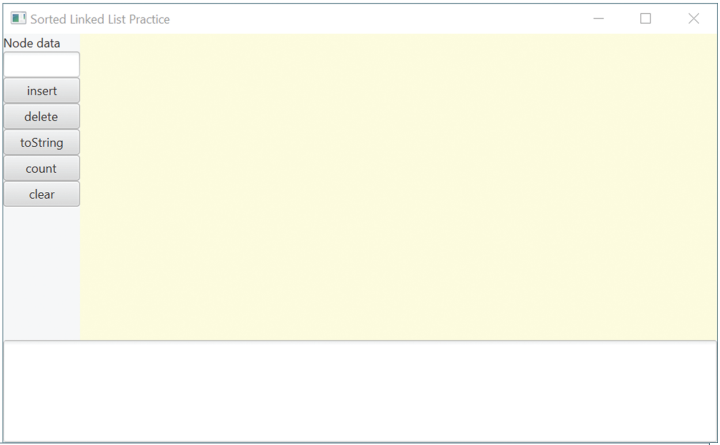
Figure 13.18 Output of Example 13.17

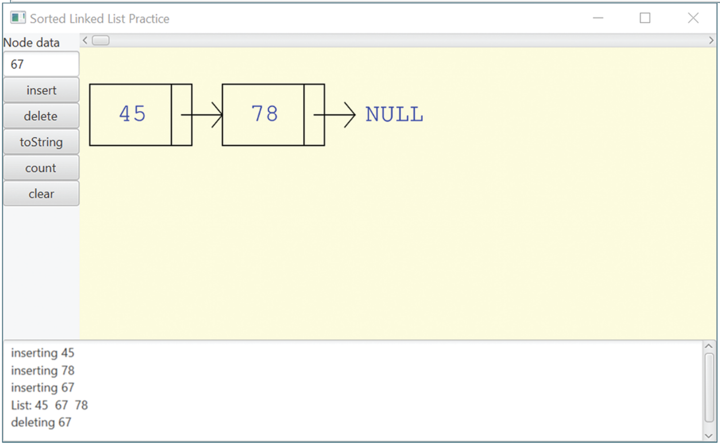
Description

CODE IN ACTION

Within the online resources, you will find a movie with a step-by-step illustration of sorted linked-list methods. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock


Sorted Linked List Practice
Node data
insert
delete
toString
count
clear


Sorted Linked List Practice
Node data
67
insert
delete
toString
count
clear
45
78
NULL

```
inserting 45
inserting 78
inserting 67
List: 45  67  78
deleting 67
```

13.9 Programming Activity 2: Writing *Insert* and *Delete* Methods for a Sorted Linked List

In this activity, you will work with a sorted linked list of integers, performing the following activity:

> Code the *insert* and *delete* methods to insert and delete nodes in a sorted linked list of *ints*.

The framework will animate your code to give you feedback on the correctness of your code. It will display the state of the sorted linked list at all times.

Instructions

Copy the contents of the Programming Activity 2 folder for this chapter from the companion website for this text onto a folder on your computer. Open the *LinkedList.java* source file. Searching for five asterisks (*****) in the source code will position you to the code section where you will add your code.

In this task, you will fill in the code inside the *insert* and *delete* methods for a sorted linked list of integers. **Example 13.18** shows the section of the *LinkedList* source code where you will add your code. This example is different from the one presented earlier in the chapter. The nodes of the linked list contain *ints*, not *Players*. The *delete* method returns a *boolean* value to indicate whether the deletion was successful. Because the client has already provided the *int* value to delete, there is no reason to return the value to the client.

Example 13.18 Location of Student Code in *LinkedList*

```
public void insert( int value )
{
  // ***** Student writes the body of this method *****
  // code the insert method of a linked list of ints
  // the int to insert in the linked list is value

  //
  // Student code starts here
  //



  //
  // End of student code, part 1
  //
}
public boolean delete( int value )
{
  // ***** Student writes the body of this method *****

  // code the delete method of a linked list of ints
  // the int to delete in the linked list is value
  // if deletion is successful, return true
  // otherwise, return false

  //
  // Student code starts here
  //

  return true; // replace this return statement
  //
  // End of student code, part 2
  //
}
```

Description

You can first code the *insert* method and run the application. Once the *insert* method works properly, you can code the *delete* method and run the application again. We have provided a dummy *return* statement in the *delete* method so that the *LinkedList.java* file will compile if only the *insert* method is coded. When you write the *delete* method, modify the dummy *return* statement to return the appropriate value.

When coding the *insert* and *delete* methods, you will need to use constructors and methods of the *Node* class. The API of the

Node class is shown in **Table 13.10**.

API of the *Node* Class

Constructors and Methods of the *Node* Class

Constructors

Class	**Constructor and argument list**
`Node`	`Node( int data )` constructs a new *Node* object whose *data* instance variable is *data*. The pointer to the next node is set to the value *null*.
`Node`	`Node( int data, Node nextNode )` constructs a new *Node* object whose *data* instance variable is *data*. The *Node* points to *nextNode*.

Methods

Return value	**Method name and argument list**
`Node`	`setNext( Node nextNode )` sets the *Node* object reference pointed to by this *Node* to *nextNode*.
`Node`	`setData( int data )` sets the *data* instance variable to data
`Node`	`getNext( )` returns an object reference to the *Node* pointed to by this *Node*.
`int`	`getData( )` returns the *data* stored in this *Node*.

Description

To test your code, compile and run the *LinkedListPractice.java* file, which contains the *main* method. When the program begins, a window will display the state of the linked list (the list is empty when we start), along with various buttons labeled “insert,” “delete,” “toString,” “count,” and “clear,” as shown in **Figure 13.19**.

Figure 13.19 Opening Window

Description

To insert or delete a value, type the integer into the text field labeled “Node Data,” then click on the “insert” or “delete” button. The application only accepts integers greater than or equal to 0 and less than or equal to 9999; it will not let you enter characters that are not digits. The main panel will visually represent the sorted linked list. The text area at the bottom will give you feedback on your operations. Close the window to exit the program.

Figure 13.20 shows the application after successively inserting 45, 67, and 78, traversing the list, then deleting 67.

Figure 13.20 Sample Window After Performing Some Operations

Description

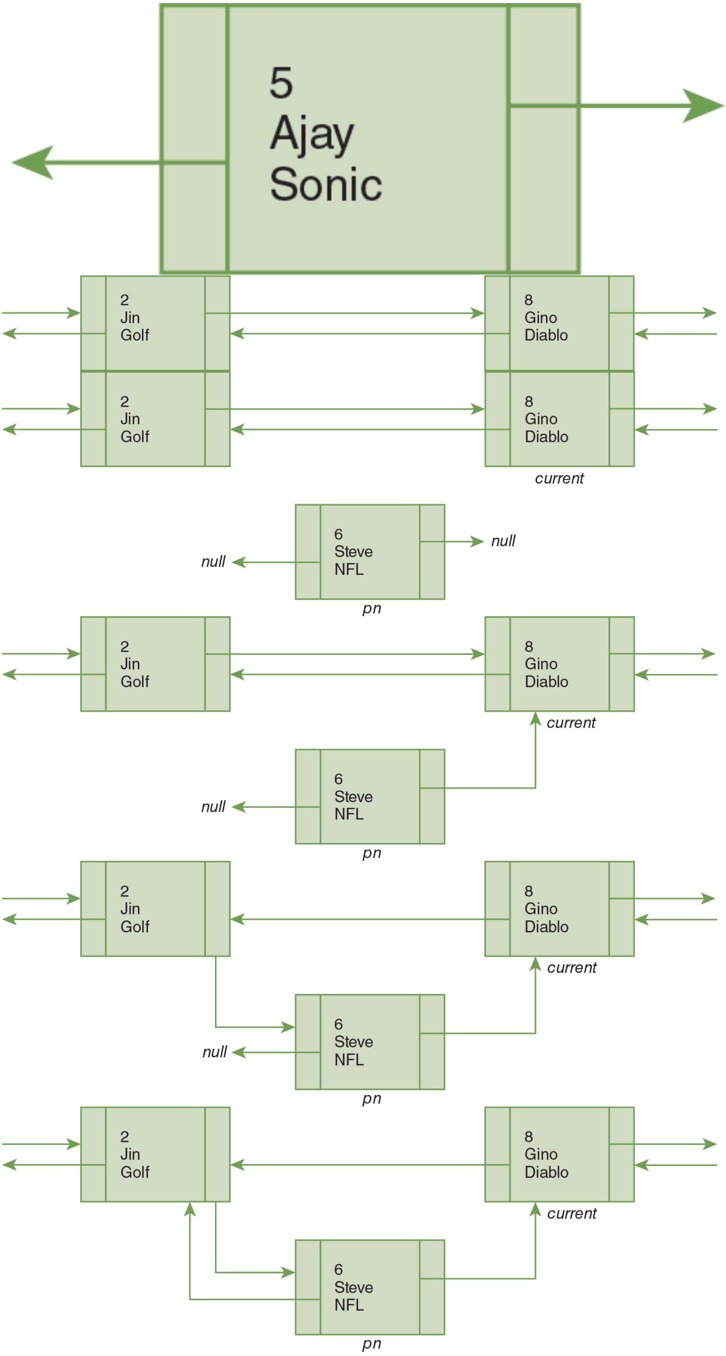
Troubleshooting

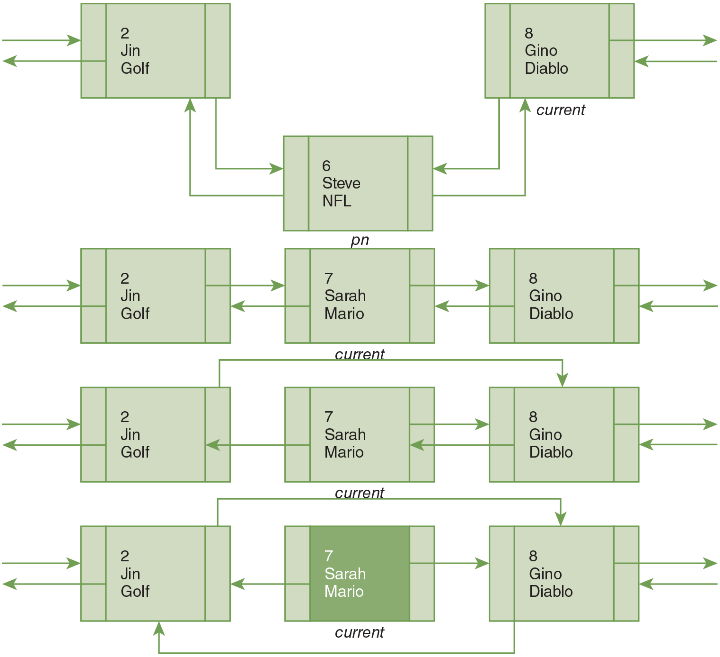
If your method implementation does not animate or animates incorrectly, check these items:

- Check the feedback in the window to see if your code gives the correct result.
- Verify that you correctly coded both cases of the *insert* method: insert at the beginning and insert in the middle of the list.
- Verify that you correctly coded all the cases of the *delete* method: fail to delete, delete at the beginning, and delete in the middle or at the end of the list.

DISCUSSION QUESTIONS

1. Explain why it is important to update *head* when inserting at the beginning of a list.
2. Explain the difference between deleting in a nonsorted list and deleting in a sorted list.


5
Ajay
Sonic
2
Jin
Golf
8
Gino
Diablo
2
Jin
Golf
8
Gino
Diablo
current
null
6
Steve
NFL
null
pn
2
Jin
Golf
8
Gino
Diablo
current
null
6
Steve
NFL
pn
2
Jin
Golf
8
Gino
Diablo
current
null
6
Steve
NFL
pn
2
Jin
Golf
8
Gino
Diablo
current
6
Steve
NFL
pn


2
Jin
Golf
8
Gino
Diablo
current
6
Steve
NFL
pn
2
Jin
Golf
7
Sarah
Mario
8
Gino
Diablo
current
2
Jin
Golf
7
Sarah
Mario
8
Gino
Diablo
current
2
Jin
Golf
7
Sarah
Mario
8
Gino
Diablo
current

13.10 Doubly Linked Lists

So far, when traversing a linked list and looking for a node containing a particular value, we have used two nodes, which we called *previous* and *current*. We kept track of the *previous* node because we had no way to go backward in the list from the *current* node.

This problem can be solved by using a **doubly linked list**, which provides two links between nodes, one forward and one backward. Using the backward link, we can now backtrack from *current* if needed. The backward link is also represented by a node object reference.

Figure 13.21 shows how we can visualize such a node. The data in the node is *5, Ajay*, and *Sonic*. The right arrow points to the next node and the left arrow points to the previous node.

Figure 13.21 A Node with Two Links

Description

In order to implement a doubly linked list, we need to modify our *Player Node* class by adding a *previous* instance variable along with its accessor and mutator methods. **Example 13.19** shows a summary of our revised *PlayerNode* class.

Example 13.19 Summary of the *PlayerNode* Class for a Doubly Linked List

```java
/* The PlayerNode class
   Anderson, Franceschi
*/

public class PlayerNode
{
  private Player player;              // the player at that node
  private PlayerNode next;            // the next PlayerNode
  private PlayerNode previous;        // the previous PlayerNode

  // constructors
  // accessors are getPlayer, getNext, getPrevious
  // mutators are setPlayer, setNext, setPrevious
}
```

Description

When inserting a node, we need to reset both forward and backward links, i.e., the *next* and *previous* instance variables. Suppose, for example, that we insert a node containing *Player p* before a node named *current*. We will illustrate only the general case, when *current* is in the middle or at the end of the doubly linked list; that is, *current* is neither *head* nor *null*.

The steps we need to perform are the following:

1. Instantiate the new node.
2. Attach the new node to *current* by setting its *next* field to *current*.
3. Attach the node before *current* to the new node by setting its *next* field to the new node.
4. Set *previous* in the new node to point to the node before *current*.
5. Set *previous* in *current* to point to the new node.
6. Add 1 to the number of items in the list.

Steps 2 and 3 set the forward links, and Steps 4 and 5 set the backward links.

Figures 13.22a to **13.22f** provide a step-by-step illustration for inserting a node in the middle of a doubly linked list. Note that we no longer need to keep a *previous* object reference, because we can get the location of the previous node from the *current* node.

Figure 13.22a Our Original Doubly Linked List

Description

Figure 13.22b Step 1: Instantiate a New Node

Description

Figure 13.22c Step 2: Set *next* in the New Node to *current*

Description

Figure 13.22d Step 3: Set *next* in the Node Before *current* to the New Node

Description

Figure 13.22e Step 4: Set *previous* in the New Node to the Node Before *current*

Description

Figure 13.22f Step 5: Set *previous* in *current* to the New Node

Description

Our code updating the links inside the *insert* method of the doubly linked list class will be the following:

```
PlayerNode pn = new PlayerNode( p );        // Step 1
pn.setNext( current );                      // Step 2
current.getPrevious( ).setNext( pn );       // Step 3
pn.setPrevious( current.getPrevious( ) );   // Step 4
current.setPrevious( pn );                  // Step 5
numberOfItems++;                            // Step 6
```

Description

The order in which these statements are executed is important. Indeed, if Step 5 were executed immediately after Step 1, we would overwrite the reference to the previous node. Then we could not access the node before *current*, and we would be unable to properly reset the links between the nodes.

Note that if *current* is either *head* (insert at the beginning) or *null* (insert at the end), the preceding code needs to be modified; that is proposed in the group project.

When deleting a node, we also need to reset all the appropriate forward and backward links. Suppose, for example, that we delete a node named *current*. We will illustrate only the general case, when *current* is in the middle of the doubly linked list. In this case, *current* is neither the *head* nor the last node in the list (since we are deleting *current*, we are assuming that *current* is not *null*); that is, there is a node after *current* in the list.

To delete a node, *current*, from the middle of a doubly linked list, we need to perform the following steps.

1. Set *next* in the node before *current* to the node after *current*.
2. Set *previous* in the node after *current* to the node before *current*.
3. Decrease the number of items by 1.

Figures 13.23a to **13.23c** give a step-by-step illustration of deleting a node.

Figure 13.23a Our Original Linked List

Description

Figure 13.23b Step 1: Set *next* in the Node Before *current* to the Node After *current*

Description

Figure 13.23c Step 2: Set *previous* in the Node After *current* to the Node Before *current*

Description

Our code updating the links inside the *delete* method of the doubly linked list class is:

```
current.getPrevious( ).setNext( current.getNext( ) );          // Step 1
current.getNext( ).setPrevious( current.getPrevious( ) );      // Step 2
numberOfItems--;                                               // Step 3
```

Description

Again, note that if *current* is either *head* or the last node in the list, the previous code would need to be modified; that is also proposed in the group project at the end of the chapter, which builds a sorted, doubly linked list.

```
Number of items in the list: 0

Number of items in the list: 1
id: 7   name: Sarah    game: Mario

Number of items in the list: 2
id: 2   name: Jin      game: Golf
id: 7   name: Sarah    game: Mario

Number of items in the list: 3
id: 5   name: Ajay     game: Sonic
id: 2   name: Jin      game: Golf
id: 7   name: Sarah    game: Mario

Player successfully deleted:
Number of items in the list: 2
id: 5   name: Ajay     game: Sonic
id: 7   name: Sarah    game: Mario

Player successfully deleted:
Number of items in the list: 1
id: 7   name: Sarah    game: Mario
```

13.11 Linked Lists Using Generic Types

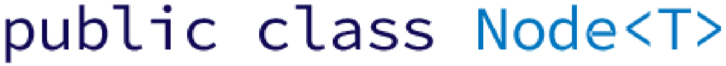
Many classes in the Java Class Library, such as the *ArrayList* class, implement generics, meaning that the data type for the data of the class can be specified by the client. User-defined classes can also implement generic types. In this section, we will build a linked list class that implements generic types, so that the client can specify the class type of the data stored in our linked list. In a linked list, the data of the item is stored in the node; thus, the data in our *Node class* will be a generic object.

The basic syntax for the header of a class that implements generics is:

```
accessModifier class ClassName<IdentifierForGenericClass>
```

Description

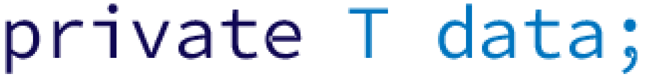
For the identifier for our generic class, we will use the uppercase letter *T*. Thus, for our *Node* class, the header will be:



```
public class Node<T>
```

Description

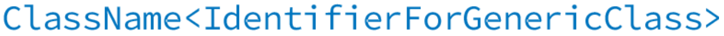
Inside the class, we can then use that identifier, here *T*, as we would use an existing or user-defined class. For example, to declare an instance variable named *data* of class *T*, we write:



```
private T data;
```

Description

In order to use an object reference of a class implementing generics, we use the following syntax:



```
ClassName<IdentifierForGenericClass>
```

Description

Thus, in order to declare an object reference of the *Node* class as a return type or a parameter for a method, we use the notation *Node<T>*.

Example 13.20 shows our *Node<T>* class.

Example 13.20 The *Node* Class Using Generics

```java
/** The Node class
*   Anderson, Franceschi
*/

public class Node<T>
{
  private T data;
  private Node<T> next;

  /**  default constructor
  *   sets data and next to null
  */
  public Node( )
  {
    data = null;
    next = null;
  }

  /** constructor
  *   @param data reference to data
  *   sets next to null
  */
  public Node( T data )
  {
    setData( data );
    next = null;
  }

  /**  accessor for data
  *   @return reference to data item
  */
  public T getData( )
  {
    return data;
  }

  /** accessor for next
  *  @return next
  */
  public Node<T> getNext( )
  {
    return next;
  }

  /**  mutator for data
  *   @param data reference to new data item
  *   @return reference to this object
  */
  public Node setData( T data )
  {
    this.data = data;
    return next;
  }

  /**  mutator for next
  *   @param reference to next Node
  *   @return reference to this object
  */
  public Node setNext( Node<T> next )
  {
    this.next = next;
    return this;
  }
}
```

Description

Example 13.21 shows our *ShellLinkedList<T>* class.

Example 13.21 The *ShellLinkedList* Class Using Generics

```java
/* The ShellLinkedList class
   Anderson, Franceschi
*/

public abstract class ShellLinkedList<T>
{
  protected Node<T> head;
  protected int numberOfItems;

  /**  constructor
  *    sets head to null and numberOfItems to 0
  */
  public ShellLinkedList( )
  {
    head = null;
    numberOfItems = 0;
  }

  /**  accessor for numberOfItems
  *   @return numberOfItems
  */
  public int getNumberOfItems( )
  {
    return numberOfItems;
  }

  /** isEmpty method
  *  @return true if no items in list; false otherwise
  */
  public boolean isEmpty( )
  {
    return ( numberOfItems == 0 );
  }

  /**  toString method
  *   @return all items in the list
  */
  @Override
  public String toString( )
  {
    String listString = "";
    Node<T> current = head;
    for ( int i = 0; i < numberOfItems; i++ )
    {
      listString += current.getData( ).toString ( ) + "\n";
      current = current.getNext( );
    }
    return listString;
  }
}
```

Description

Our *GenericLinkedList* class, shown in **Example 13.22**, implements the same methods as our *PlayerLinkedList* from earlier in the chapter. The only differences in the code are the class header and the declaration and instantiation of *Node* variables.

Example 13.22 The *GenericLinkedList* Class with Generics

```java
/* The GenericLinkedList class
   Anderson, Franceschi
*/

public class GenericLinkedList<T> extends ShellLinkedList<T>
{
  // head and numberOfItems are inherited instance variables

  /**  constructor
  *   calls constructor of ShellLinkedList
  */
  public GenericLinkedList( )
  {
    super( );
  }

  /** insert method
  *  @param     item   T object to insert
  */
  public void insert( T item )
  {
    // insert as head
    Node<T> nd = new Node<T>( item );
    nd.setNext( head );
    head = nd;
    numberOfItems++;
  }

  /** delete method
  *   @param    item   T object to delete
  *   @return   true if the deletion was successful, false otherwise
  */
  public boolean delete( T item )
  {
    Node<T> current = head;
    Node<T> previous = null;
    while ( current != null
            &&  ! ( item.equals ( current.getData( ) ) ) )
    {
      previous = current;
      current = current.getNext( );
    }

    if ( current == null ) // not found
         return false;
    else
    {
      if ( current == head )
        head = head.getNext( ); // delete head
      else
        previous.setNext ( current.getNext( ) );

      numberOfItems--;
      return true;
    }
  }
}
```

Description

The *insert* method, coded at lines 17–27, is very similar to the *insert* method of the *PlayerLinkedList* class. Instead of a *Player* reference, its parameter *item* is a *T* reference, where *T* is a generic class. It also inserts *item* at the beginning of the list. At line 23, we declare and instantiate a *Node<T>* object reference, which is then connected to the list. The rest of the *insert* method is identical to the code in our *insert* method of a nongeneric linked-list class.

Because we do not know in advance what type of object our class will be instantiated with, we implemented our *delete* method differently. We cannot delete an item based on the value of one of its fields because we do not know what the fields of that item are, since that item is a generic object. Thus, the parameter of our *delete* method is a generic object of the same type as the items in the list. There is no need to return an item if we find it and can delete it because we already have that item as the parameter of the method. For that reason, our *delete* method returns a *boolean* value: *true* if we were able to delete the parameter item, *false* otherwise. In order to compare *item* with the items in the list, we call the *equals* method at line 38, inherited by any class from the *Object* class, and which will need to be overwritten in the class the client specifies as the type for the linked list.

Now that we have defined and implemented our linked list class storing generic objects, how do we use it in a client class? We use the same syntax as we would using an existing Java class implementing generics. In fact, in the *GenericLinkedList* class, we used the *Node* class, which implements generics.

Example 13.23 shows a client class using the *GenericLinkedList* class.

Example 13.23 The *LinkedListTest* Class

```
 1 /* The LinkedListTest class
 2    Anderson, Franceschi
 3 */
 4
 5 public class LinkedListTest
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     Player p1 = new Player( 7,"Sarah","Mario" );
10     Player p2 = new Player( 2,"Jin","Golf" );
11     Player p3 = new Player( 5,"Ajay","Sonic" );
12
13     // construct empty LinkedList of Player objects
14     GenericLinkedList<Player>players = new GenericLinkedList<Player>( );
15     System.out.println( "Number of items in the list: "
16      + players.getNumberOfItems( ) + "\n" + players.toString( ) );
17
18     players.insert( p1 ); // insert in empty list
19     System.out.println( "Number of items in the list: "
20      + players.getNumberOfItems( ) + "\n" + players.toString( ) );
21
22     players.insert( p2 ); // insert in list of one item
23     System.out.println( "Number of items in the list: "
24      + players.getNumberOfItems( ) + "\n" + players.toString( ) );
25
26     players.insert( p3 ); // insert in list of two items
27     System.out.println( "Number of items in the list: "
28      + players.getNumberOfItems( ) + "\n" + players.toString( ) );
29
30     if ( players.delete( p2 ) ) // delete in the middle
31         System.out.println( "Player successfully deleted: " );
32     System.out.println( "Number of items in the list: "
33      + players.getNumberOfItems( ) + "\n" + players.toString( ) );
34
35     if ( players.delete( p3 ) ) // delete at the beginning
36        System.out.println( "player successfully deleted: " );
37     System.out.println( "Number of items in the list: "
38      + players.getNumberOfItems( ) + "\n" + players.toString( ) );
39   }
40 }
```

Description

The only statement that is specific to the generic character of the *GenericLinkedList* is at line 14 when we declare and instantiate an object reference of *GenericLinkedList*. If we wanted to declare and instantiate a list containing *Integer* objects, we would have written:

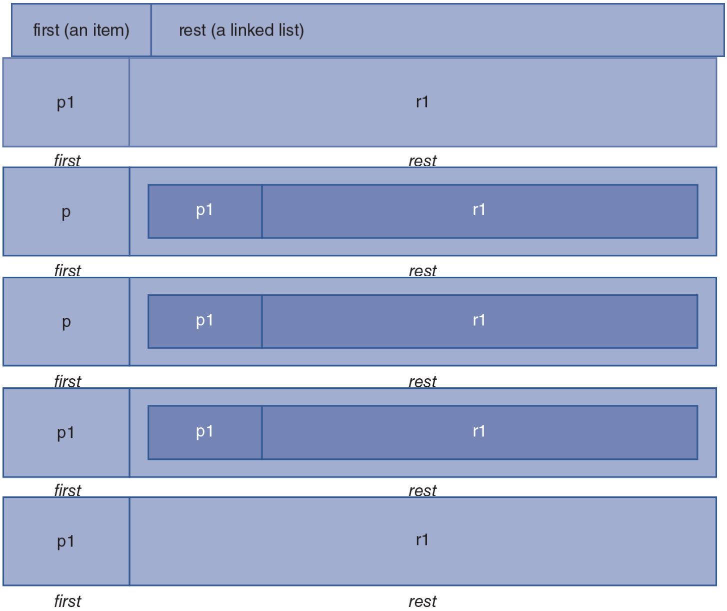
```
GenericLinkedList<Integer> numbers = new GenericLinkedList<Integer>( );
```

Description

Figure 13.24 shows the output of Example 13.23.

Figure 13.24 Output of Example 13.23

Description


first (an item)
rest (a linked list)
p1
r1
first
rest
p
p1
r1
first
rest
p
p1
r1
first
rest
p1
p1
r1
first
rest
p1
r1
first
rest

```
The list is
empty

Inserting id: 7         name: Sarah game: Mario
The list is
id: 7    name: Sarah    game: Mario

Inserting id: 2         name: Jin  game: Golf
The list is
id: 2    name: Jin      game: Golf
id: 7    name: Sarah    game: Mario

Inserting id: 5         name: Ajay  game: Sonic
The list is
id: 5    name: Ajay     game: Sonic
id: 2    name: Jin      game: Golf
id: 7    name: Sarah    game: Mario

Player deleted: id: 2   name: Jin  game: Golf
The list is
id: 5    name: Ajay     game: Sonic
id: 7    name: Sarah    game: Mario

Player deleted: id: 7   name: Sarah game: Mario
The list is
id: 5    name: Ajay     game: Sonic

The list is
id: 5    name: Ajay     game: Sonic

Player deleted: id: 5   name: Ajay  game: Sonic
The list is
empty

The list is
empty
```

13.12 Recursively Defined Linked Lists

A linked list can be defined recursively. A recursively defined linked list is made up of two items:

- *first*, an item, which is the first item in the linked list
- *rest*, a linked list, which consists of the rest of the linked list

Figure 13.25 shows a representation of a recursively defined linked list.

Figure 13.25 A Recursively Defined Linked List

Description

In our recursively defined linked list, we have two instance variables: the item *first* and the linked list *rest*. Because we can access the rest of the list through the *rest* instance variable, we do not need a node class.

In designing our class encapsulating a recursive linked list of generic objects, we will limit ourselves to an unsorted linked list. We will insert at the beginning of the list. When we delete, we will attempt to delete and return an object that matches a parameter object. When we cannot delete, we will return *false*.

Table 13.11 shows the APIs of the *insert* and *delete* methods.

***RecursiveLinkedList* Methods**

Methods of the *RecursiveLinkedList* Class

Return value	Method name and argument list
`void`	`insert( T item )` inserts *item* at the beginning of the list.
`boolean`	`delete( T item )` removes the first object of the list that matches *item* and returns *true*. If there is no such object in the list, the method returns *false*.

Description

After we insert, *first* will hold the item inserted, and *rest* will hold the original list. **Figures 13.26a** and **13.26b** show a recursively defined linked list before and after inserting a *Player* named *p*. In the figures, *p1* represents the current first item, and *r1* represents the rest of the list before the insertion. The *insert* method is not recursive.

Figure 13.26a The List Before Inserting *Player p*

Description

Figure 13.26b The List After Inserting *Player p*

Description

The *delete* method is recursive. We have three base cases:

- The list is empty.
- The element to delete is the first item of the list.
- The element to delete is not the first item of the list and the rest of the list is empty.

In the general case, we try to delete the element from the rest of the list.

If the list is empty (the first base case), we will return *false*. If the list is not empty, we will look at *first* and check to see if it matches the parameter *item*. If it does (the second base case), we will delete *first*, and *rest* will become our list. If it does not, then we will attempt to delete inside *rest*. If rest is *null*, we cannot delete (the third base case) and we will return *false*. If rest is not *null*, we will make a recursive call to the *delete* method with *rest* (the general case).

More generally, we want to do the following:

- If the list is empty (base case #1), the method returns.
- Process *first*, that is, the first element in the list (base case #2); the method may or may not return at that point.
- If *rest* is *null*—that is, the list has only 1 item (base case #3)—the method returns.
- If *rest* is not *null*, make a recursive call on *rest*.

Example 13.24 shows our *RecursiveLinkedList* class. Because of its recursive design, the *RecursiveLinkedList* class does not extend the *ShellLinkedList* class.

Example 13.24 The *RecursiveLinkedList* Class

```java
/* The RecursiveLinkedList class
   Anderson, Franceschi
*/

public class RecursiveLinkedList<T>
{
  private T first;
  private RecursiveLinkedList<T> rest;

  public RecursiveLinkedList( )
  {
    first = null;
    rest = null;
  }

  /** insert method
  *   @param    item object to insert at beginning of list
  */
  public void insert( T item )
  {
    if ( isEmpty ( ) ) // is list empty?
      first = item;
    else
    {
      RecursiveLinkedList<T> tempList =
                 new RecursiveLinkedList<T>( );
      tempList.first = first;
      tempList.rest = rest;
      first = item;
      rest = tempList;
    }
  }

  /** delete method
  *   @param    item    the T object to delete
  *   @return   true if item is deleted, false otherwise
  */
  public boolean delete( T item )
  {
    if ( isEmpty( ) )                   // is list empty?
      return false;
    else if ( first.equals( item ) )  // found it
    {
      T temp = first;
      if ( rest == null )
          first = null;
      else       // rest not null
      {
          first = rest.first;
          rest = rest.rest;
      }
      return true;
    }
    else if ( rest == null )
        return false;
    else                        // try to delete in rest
        return rest.delete( item );
   }

   /** isEmpty  method
  *   @return   true if the list has no elements;
  *             false, otherwise
  */
  public boolean isEmpty( )
  {
    return ( first == null );
  }

  /** toString method
  *  @return a String listing the elements in the list
  */
  @Override
  public String toString( )
  {
    String listString = "";
    if ( first != null )
    {
      listString = first.toString( ) + "\n";
      if ( rest != null )
        listString += rest.toString( );
    }
    return listString;
  }
}
```

Description

We declare the two instance variables at lines 7–8: *first* represents the first *T* object in the list, and *rest* represents the rest of the list, which is a *RecursiveLinkedList* object reference itself. We coded the default constructor, which constructs an empty list, at lines 10–14.

We coded the *insert* method at lines 16–32. After insertion, *first* will be the method's *T* parameter *item*, and *rest* will be the list before we inserted *item*. We begin by testing if the list is empty by calling the *isEmpty* method (defined at lines 60 to 67), which returns *true* if *first* is *null*. If the list is empty, we assign *item* to *first* at line 22. If *first* is not *null*, we copy the current list into a new list at lines 25–28. We instantiate a temporary list, *tempList*. We then assign *first* to the *first* instance variable of *tempList* and *rest* to the *rest* instance variable of *tempList*. At that point, we have copied the current list into *tempList*. Now we can insert the new item into the first position (line 29) and make *tempList* the rest of the list (line 30).

The recursive *delete* method (lines 34–58) takes the *T* parameter *item*. If the list is empty (line 40), we return *false*. If the list is not empty, then *first* is not *null*, and we can call the *equals* method on *first*. More generally, when processing a recursively designed list, not testing for all the base case conditions could result in a *NullPointerException*.

If the list is not empty and *first* is equal to *item* (line 42), we do the necessary bookkeeping on the list to delete the first element at lines 44–51 before returning *true* at line 52. In order to delete the first element of the list, we need to update *first* and *rest*. *First* will be assigned the first element of *rest*. However, *rest* could be *null*, in which case *rest* does not have a first element. Thus, we test if *rest* is *null* at line 45. If it is, the list is now empty, so we assign *null* to *first* at line 46. If *rest* is not *null*, we assign the first element of *rest* to *first* at line 49, and we assign the *rest* of *rest* to *rest* at line 50.

Figures 13.27a to **13.27c** show the list before deleting *Player p*, after line 49 is executed, and after line 50 is executed, when *Player p* has been deleted from the list.

Figure 13.27a The List Before Deleting *Player p*

Description

Figure 13.27b The List After *first* Is Assigned *first* of *rest*

Description

Figure 13.27c The List After *rest* of *rest* Is Assigned to *rest; Player p* Has Been Deleted

Description

Finally, if the list is not *null* and the *first* is not equal to *item*, we skip to line 54, where we test if *rest* is *null*. If it is, we cannot delete and return *false*. If *rest* is not *null*, we make the recursive call to try to delete from *rest* at line 57.

We coded our *toString* method at lines 69–83. This method is also recursive. If the list is empty, it returns the empty *String*. If the list is not empty, we assign the contents of *first* to the temporary variable *listString* at line 78. Note that the class the client specifies as the type of the *RecursiveLinkedList* will need to provide an overriding *toString* method. We then need to traverse *rest* in order to add its contents to *listString*. But *rest* could be *null*, in which case we are finished traversing the list. So if *rest* is not *null* (line 79), we traverse *rest* at line 80 by making the recursive call:

```
rest.toString( )
```

Description

That recursive call returns a *String* representing the contents of *rest*; we concatenate that *String* to *listString* at line 80 before returning *listString* at line 82.

Example 13.25 shows how to use our *RecursiveLinkedList* class in a client program.

Example 13.25 The *RecursiveLinkedListTest* Class

```java
/* The RecursiveLinkedListTest class
   Anderson, Franceschi
*/

public class RecursiveLinkedListTest
{
  public static void main( String [ ] args )
  {
    Player p1 = new Player( 7,"Sarah","Mario" );
    Player p2 = new Player( 2,"Jin","Golf" );
    Player p3 = new Player( 5,"Ajay","Sonic" );

    RecursiveLinkedList<Player> players =
        new RecursiveLinkedList<Player>( );
    System.out.println( "The list is\n"
        + ( players.isEmpty( ) ? "empty\n" : players.toString( ) ) );

    players.insert( p1 );
    System.out.println( "Inserting " + p1 );
    System.out.println( "The list is\n"
        + ( players.isEmpty( ) ? "empty\n" : players.toString( ) ) );

    players.insert( p2 );
    System.out.println( "Inserting " + p2 );
    System.out.println( "The list is\n"
        + ( players.isEmpty( ) ? "empty\n" : players.toString( ) ) );

     players.insert( p3 );
     System.out.println( "Inserting " + p3 );
     System.out.println( "The list is\n"
        + ( players.isEmpty( ) ? "empty\n" : players.toString( ) ) );

     if ( players.delete( p2 ) )  // delete in middle of list
       System.out.println( "Player deleted: " + p2 );
     System.out.println( "The list is\n"
         + ( players.isEmpty( ) ? "empty\n" : players.toString( ) ) );

     if ( players.delete( p1 ) )  // delete at end of the list
       System.out.println( "Player deleted: " + p1 );
     System.out.println( "The list is\n"
         + ( players.isEmpty( ) ? "empty\n" : players.toString( ) ) );

     if ( players.delete( p1 ) )  // attempt to delete will fail
       System.out.println( "The list deleted: " + p1 );
     System.out.println( "The list is\n"
         + ( players.isEmpty( ) ? "empty\n" : players.toString( ) ) );

     if ( players.delete( p3 ) )  // delete only Player in list
       System.out.println( "Player deleted: " + p3 );
     System.out.println( "The list is\n"
         + ( players.isEmpty( ) ? "empty\n" : players.toString( ) ) );

     if ( players.delete( p3 ) )  // try to delete from empty list
       System.out.println( "Player deleted: " + p3 );
     System.out.println( "The list is\n"
         + ( players.isEmpty( ) ? "empty\n" : players.toString( ) ) );
  }
}
```

Description

COMMON ERROR TRAP

When processing a recursively defined list, not testing for all the base case conditions can eventually result in a *NullPointerException* at run time.

In Example 13.25, we again instantiate our usual three *Player* object references *p1*, *p2*, and *p3* at lines 9–11. We instantiate the *RecursiveLinkedList players* at lines 13–14. This example tests the following operations:

- inserting in an empty list (line 18)
- inserting in a list of one element (line 23)
- inserting in a list of two elements (line 28)
- deleting an element in the middle of the list (line 33)
- deleting an element at the end of the list (line 38)
- failing to delete from a non-empty list (line 43)
- deleting the only element in the list (line 48)
- failing to delete from an empty list (line 53)

SOFTWARE ENGINEERING TIP

When a class is defined recursively, think in terms of implementing recursive methods.

Figure 13.28 shows the output of Example 13.25.

Figure 13.28 Output of Example 13.25

Description

with these end-of-chapter questions

13.14.1	Multiple Choice Exercises Questions 1, 8, 9, 10, 11, 12, 13
13.14.4	Identifying Errors in Code Questions 35, 36, 37, 38
13.14.6	Write a Short Program Questions 47, 49, 51, 54, 55, 56, 57, 58
13.14.8	Technical Writing

© Lois GoBe/Shutterstock.

CHAPTER REVIEW

13.13 Chapter Summary

- A data structure is a mechanism for organizing the data a program stores in memory.
- A linked list is a data structure consisting of nodes linked together like a chain.
- Typical instance variables for a node are an object reference to the data stored at the node, and a node reference, which points to the next node in the list.
- Because each node has a reference to the next node as an instance variable, a linked list needs only one instance variable, its first node, which is usually called *head*. Often, for convenience, we also include an instance variable representing the number of items in the list.
- A linked list can be expanded one node at a time, therefore optimizing memory use.
- A stack is a data structure organized as last in, first out.
- A queue is a data structure organized as first in, first out.
- A linked list can be used to represent a stack. In that case, we push onto the stack by inserting an item at the beginning of the list. We pop by deleting the first item of the list.
- A linked list can also be used to represent a queue. In that case, we enqueue by inserting at the end of the list. We

dequeue by deleting the first item of the list. Because we insert at the end of the list, it is useful to have an instance variable representing the last node in the list, often called *tail*.

- A stack can also be represented by an array, if we know in advance the maximum number of items that will be stored on the stack at one time. An instance variable called *top* represents the index of the last array element pushed onto the stack. We pop, or delete, that element, unless the stack is empty. We push, or insert, onto the stack a new element at index (*top* + *1*), unless the stack is full.
- A queue can also be represented by an array, if we know in advance the maximum number of items that will be stored in the queue at one time. A circular array is usually implemented for the queue. Two instance variables called *front* and *back* represent the indexes of the first and last element inserted in the queue. We dequeue, or delete, the element at index *front* unless the queue is empty. We enqueue, or insert, a new element at index (*back* + *1*% *QUEUE_SIZE*), unless the queue is full.
- In a class encapsulating a data structure, a method returning an object can *throw* an exception if we cannot return or cannot find the object. Indeed, when the object we are looking for is not found, it is preferable to *throw* an exception rather than to return *null*.
- A linked list can be sorted in ascending or descending order. One of the instance variables of the list objects is used as the key to sort the list elements. The *insert* method finds the appropriate location to insert an item so that the list remains sorted.
- A variation of the linked list includes a doubly linked list. In this case, each node contains three instance variables: an object representing the data, a node reference representing the next node, and another node reference representing the previous node. The latter enables us to backtrack in the list, if we need to, whereas in a singly linked list, we can traverse the list in a forward direction only. However, implementing such a list is more difficult; each method, in particular *insert* and *delete*, involves more operations to

maintain these double links between nodes.

- Linked lists can implement generic types; when such a linked list is instantiated, the client specifies the class of the items for the list.
- Linked lists can also be recursively defined. A recursively defined linked list is made up of two elements: *first*, which is the first item in the linked list, and *rest*, a linked list that consists of the rest of the linked list.

13.14 Exercises, Problems, and Projects

13.14.1 Multiple Choice Exercises

1. What is an advantage of linked lists over arrays?

    - Linked lists are easily expanded.
    - Linked lists are limited in size.
    - Linked lists can store objects, whereas arrays are limited to primitive data types.

5. How is a stack organized?

    - FIFO
    - LIFO
    - Items are sorted in ascending order.
    - Items are sorted in descending order.

10. How is a queue organized?

    - FIFO
    - LIFO
    - Items are sorted in ascending order.
    - Items are sorted in descending order.

15. The following linked list represents a stack. If we pop once from the stack, what item is popped?

    (*7, Ajay, NFL*) → (*3, Sarah, Mario*) → (*9, Jin, Golf*)
    head
    → (*5, Joe, Sonic*) → *null*

    - (*7, Ajay, NFL*)
    - (*3, Sarah, Mario*)
    - (*9, Jin, Golf*)
    - (*5, Joe, Sonic*)

20. The linked list that follows represents a stack. After we push the player (*5, Joe, Sonic*) onto the stack, what are the first and last items on the stack?

(*7, Ajay, NFL*) → (*3, Sarah, Mario*) → (*9, Jin, Golf*) → *null*
head

- (*7, Ajay, NFL*) and (*9, Jin, Golf*)
- (*5, Joe, Sonic*) and (*9, Jin, Golf*)
- (*3, Sarah, Mario*) and (*5, Joe, Sonic*)
- (*7, Ajay, NFL*) and (*5, Joe, Sonic*)

25. The linked list that follows represents a queue. If we dequeue once, what item is dequeued?

(*7, Ajay, NFL*) → (*3, Sarah, Mario*) → (*9, Jin, Golf*) →
head
(*5, Joe, Sonic*) → *null*
tail

- (*7, Ajay, NFL*)
- (*3, Sarah, Mario*)
- (*9, Jin, Golf*)
- (*5, Joe, Sonic*)

30. The linked list that follows represents a queue. After we enqueue the player (*5, Joe, Sonic*), what are now the first and last items on the queue?

(*7, Ajay, NFL*) → (*3, Sarah, Mario*) → (*9, Jin, Golf*) → *null*
head *tail*

- (*7, Ajay, NFL*) and (*9, Jin, Golf*)
- (*5, Joe, Sonic*) and (*9, Jin, Golf*)
- (*3, Sarah, Mario*) and (*5, Joe, Sonic*)
- (*7, Ajay, NFL*) and (*5, Joe, Sonic*)

35. The diagram that follows shows the current state of a stack

represented by an array of 50 integers. After pushing 36 and 62 onto the stack and then popping once, what will be the value of *top*, and what element will be stored at index *top*?

Index	**Item stored**
47 (top)	28
46	98
…	…
3	17
2	12
1	20
0	45

- *top* is 47 and the element at index *top* is 28
- *top* is 49 and the element at index *top* is 62
- *top* is 48 and the element at index *top* is 36
- *top* is 49 and the element at index *top* is 20

40. The diagram that follows shows the current state of a stack represented by an array of 50 integers. After pushing 36, 88, and 62 onto the stack and popping three times from the stack, what will be the value of *top* and what element will be stored at index *top*?

Index	**Item stored**
47 (top)	28
46	98
…	…
3	17
2	12
1	20
0	**45**

- *top* is 49 and the element at index *top* is 62
- *top* is 47 and the element at index *top* is 28
- *top* is 46 and the element at index *top* is 98
- *top* is 50 and the element at index *top* is 17

45. The diagram that follows shows the current state of a queue represented by a circular array of 8 integers. After enqueuing 36 and 62, and dequeuing once, what are the values of *front* and *back*, and what elements are stored at indexes *front* and *back*?

Index	**Item stored**
7	
6 (back)	28
5	97
4	25
3	54
2 (front)	12
1	
0	

- *front* = 0, stores 62; *back* = 5, stores 97
- *front* = 3, stores 54; *back* = 0, stores 62
- *front* = 3, stores 54; *back* = 8, stores 62
- *front* = 1, stores 36; *back* = 6, stores 28

50. The diagram that follows shows the current state of a queue represented by a circular array of 8 integers. After enqueuing 36, 100, 83, 77, and 62, what are the values of *front* and *back*, and what elements are stored at indexes *front* and *back*?

Index	**Item stored**
7	
6 (back)	28
5	97
4	25
3	54
2 (front)	12
1	
0	

- *front* = 2, stores 12; *back* = 11, stores 62
- *front* = 2, stores 12; *back* = 3, stores 62
- *front* = 3, stores 62; *back* = 6, stores 28

- *front* = 2, stores 12; *back* = 1, stores 83

55. The diagram that follows shows the current state of a queue represented by a circular array of 8 integers. After dequeuing 5 times, what are the values of *front* and *back*, and what elements are stored at indexes *front* and *back*?

Index	**Item stored**
6 (back)	28
5	97
4	25
3	54
2 (front)	12
1	
0	

- *front* = 7; *back* = 6; the queue is empty
- *front* = 2; *back* = 1; the queue is empty
- *front* = 2; *back* = 6; the queue is empty

59. The diagram that follows shows the current state of a queue represented by a circular array of 8 integers. After dequeuing 8 times, what are the values of *front* and *back*, and what elements are stored at indexes *front* and *back*?

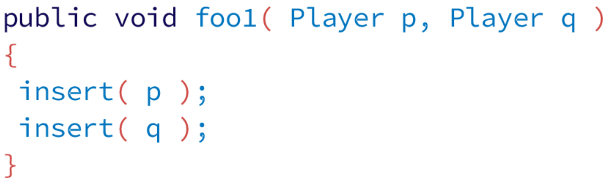
Index	**Item stored**
7	
6 (back)	28
5	97
4	25
3	54
2 (front)	12
1	
0	

- *front* = 7; *back* = 6; the queue is empty
- *front* = 2; *back* = 1; the queue is empty
- *front* = 2; *back* = 6; the queue is empty
- *front* = 6; *back* = 2; the queue is empty

13.14.2 Reading and Understanding Code

For Questions 14 to 21, consider the following classes from this chapter: *Player*, *PlayerNode*, and *PlayerLinkedList*.

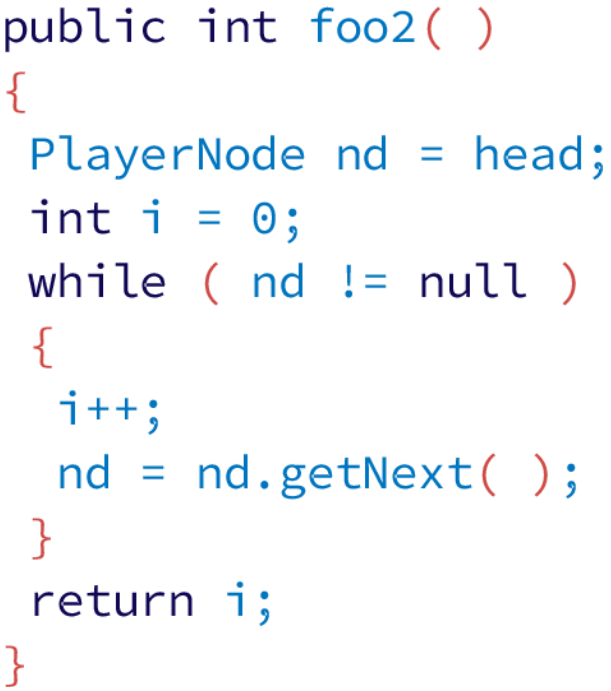
1. What does this method of the *PlayerLinkedList* class do?



```
public void foo1( Player p, Player q )
{
 insert( p );
 insert( q );
}
```

Description

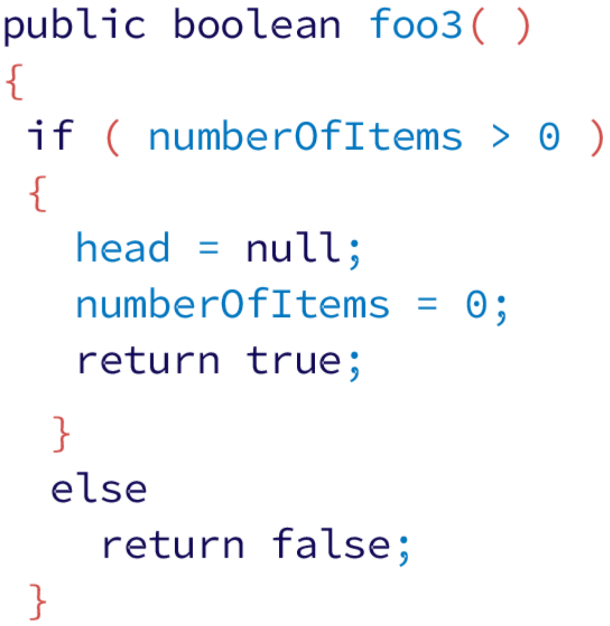
2. What does this method of the *PlayerLinkedList* class do?



```
public int foo2( )
{
 PlayerNode nd = head;
 int i = 0;
 while ( nd != null )
 {
  i++;
  nd = nd.getNext( );
 }
 return i;
}
```

Description

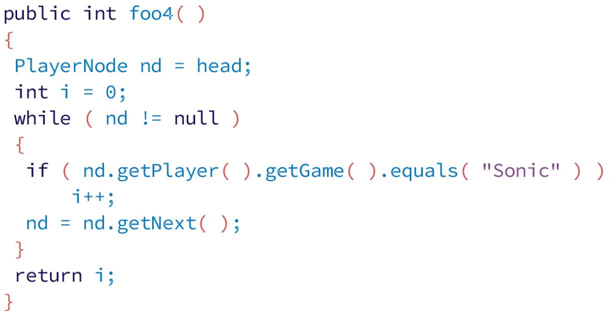
3. What does this method of the *PlayerLinkedList* class do?



```
public boolean foo3( )
{
 if ( numberOfItems > 0 )
 {
    head = null;
    numberOfItems = 0;
    return true;
  }
  else
     return false;
 }
```

Description

4. What does this method of the *PlayerLinkedList* class do?



```
public int foo4( )
{
 PlayerNode nd = head;
 int i = 0;
 while ( nd != null )
 {
  if ( nd.getPlayer( ).getGame( ).equals( "Sonic" ) )
      i++;
  nd = nd.getNext( );
 }
 return i;
}
```

Description

5. What does this method of the *PlayerLinkedList* class do?

```
public boolean foo5( int i )
{
 PlayerNode nd = head;
 while ( nd != null )
 {
  if ( nd.getPlayer( ).getID( ) == i )
     return true;
  nd = nd.getNext( );
 }
 return false;
}
```

Description

6. What does this method of the *PlayerLinkedList* class do?

```
public void foo6( )
{
 PlayerNode nd = head;
 while ( nd != null )
 {
   if ( nd.getPlayer( ).getGame( ).equals( "Diablo" ) )
     System.out.println( nd.getPlayer( ).toString( ) );
   nd = nd.getNext( );
 }
}
```

Description

7. What does this method of the *PlayerLinkedList* class do?

```
public void foo7( Player p )
{
    if ( numberOfItems == 0 )
      System.out.println( "Do nothing" );
    else
    {
      PlayerNode pn = new PlayerNode( p );
      pn.setNext( head.getNext( ) );
      head.setNext( pn );
      numberOfItems++;
   }
    }
```

Description

8. What does this method of the *PlayerLinkedList* class do?

```
public boolean foo8( )
{
 if ( numberOfItems <= 2 )
   return false;
 else
 {
   head.setNext( ( head.getNext( ) ).getNext( ) );
   numberOfItems--;
   return true;
 }
}
```

Description

13.14.3 Fill In the Code

1. Consider the following state of a linked list of *Player* items.

   → (*7*, *Ajay*, *NFL*) → (*3*, *Sarah*, *Mario*) → (*9*, *Jin*, *Golf*) → (*5*, *Joe*, *Sonic*)

   previous

   As indicated, *previous* is the *PlayerNode* whose player is (*7*, *Ajay*, *NFL*).

   Write the code to modify the list so that (*9*, *Jin*, *Golf*) has been deleted.

   → (*7*, *Ajay*, *NFL*) → (*3*, *Sarah*, *Mario*) → (*5*, *Joe*, *Sonic*) →

   ```
   // your code goes here
   ```

2. Consider the following state of a linked list of *Player* items.

   → (*7*, *Ajay*, *NFL*) → (*3*, *Sarah*, *Mario*) → (*9*, *Jin*, *Golf*) → (*5*, *Joe*, *Sonic*)

   previous

   As indicated, *previous* is the *PlayerNode* whose player is (*7*, *Ajay*, *NFL*).

   Write the code to modify the list so that the two items in the middle have been deleted.

   → (*7*, *Ajay*, *NFL*) → (*5*, *Joe*, *Sonic*) →

   ```
   // your code goes here
   ```

3. Consider the following state of a linked list of Player items.

→ (*7*, *Ajay*, *NFL*) → (*3*, *Sarah*, *Mario*) → (*9*, *Jin*, *Golf*) → (*5*, *Joe*, *Sonic*)

previous *current*

As indicated, *previous* is the *PlayerNode* whose player is (*7*, *Ajay*, *NFL*) and *current* is the *PlayerNode* whose player is (*3*, *Sarah*, *Mario*). Write the code to modify the list so that the two nodes in the middle have been swapped as shown here. (You need to swap the actual nodes, rather than modify their respective data.)

→ (*7*, *Ajay*, *NFL*) → (*9*, *Jin*, *Golf*) → (*3*, *Sarah*, *Mario*) → (*5*, *Joe*, *Sonic*)

```
// your code goes here
```

For Questions 25 to 28, consider the *LLNode* class that follows, representing a node with a *char* instance variable, representing a grade (A, B, C, D, or F):

```
public class LLNode
{
 private char grade;
 private LLNode next;
     // constructors and methods here
}
```

Description

4. Code the overloaded constructor with one parameter, a *char*.

```
// your code goes here
```

Description

5. Code the overloaded constructor with two parameters.

```
// your code goes here
```

Description

6. Code the accessors for the class.

```
// your code goes here
```

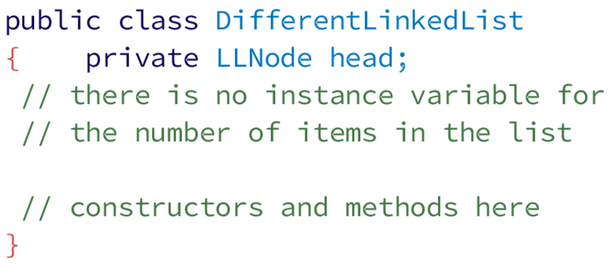
Description

7. Code the mutators for the class.

```
// your code goes here
```

Description

For Questions 29 to 31, consider the following *DifferentLinkedList* class, using the *LLNode* class from Questions 25 to 28 (assume that the *LLNode* class has all appropriate accessors, mutators, and other methods).



```
public class DifferentLinkedList
{    private LLNode head;
 // there is no instance variable for
 // the number of items in the list

 // constructors and methods here
}
```

Description

8. Code a method that returns *true* if the list is empty; *false* otherwise.

```
// your code goes here
```

Description

9. Code a method that returns *true* if the list contains at least one item; *false* otherwise.

```
// your code goes here
```

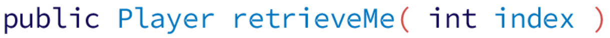
Description

10. Code a method that returns the number of items in the list.

```
// your code goes here
```

Description

11. Consider a method of class *PlayerLinkedList* with the following header:



```
public Player retrieveMe( int index )
```

Description

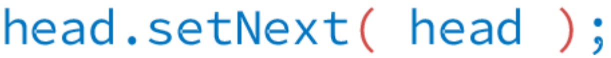
Write a few statements showing how you would call that method from a client program.

```
// your code goes here
```

Description

13.14.4 Identifying Errors in Code

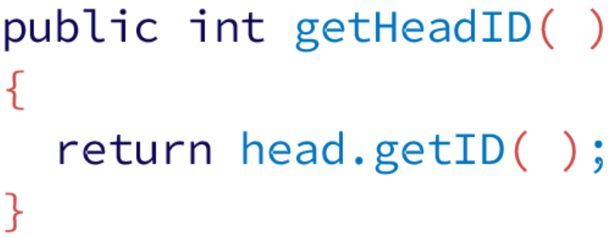
1. What would happen if you execute the following code just before traversing a linked list?



```
head.setNext( head );
```

Description

2. Suppose we have coded the following method in the *PlayerLinkedList* class. Where is the error?



```
public int getHeadID( )
{
  return head.getID( );
}
```

Description

3. Suppose we modify the code of the *push* method in the

StackArray class as follows (that is, without incrementing *top*). What type of problem could that method create?

```
public boolean push( Player p )
{
  if ( !isFull( ) ) // is there room to insert?
  {
     stack[ top ] = p;
     return true;
  }
  else
     return false;
}
```

Description

4. Suppose we modify the code of the *pop* method in the *StackArray* class as follows (that is, without decrementing *top*). What type of problem could that method create?

```
public Player pop( ) throws DataStructureException
{
 if ( !isEmpty( ) ) // is there an item to delete?
  return ( stack[top] );
 else
  throw new DataStructureException
          ( "Stack empty: cannot pop" );
}
```

Description

5. Suppose we modify the code of the *enqueue* method in the *QueueArray* class as follows (that is, without incrementing the number of items). What type of problem could that method create?

```
public boolean enqueue( Player p )
{
  if ( !isFull( ) )
  {
   queue[( back + 1 ) % QUEUE_SIZE] = p;
   back = ( back + 1 ) % QUEUE_SIZE;
   return true;
  }
  else
   return false;
}
```

Description

6. Suppose we modify the code of the *dequeue* method in the *QueueArray* class as follows (that is, with a change in the expression computing the index in the *return* statement). Where is the error?

```
public Player dequeue( ) throws DataStructureException
{
 if ( !isEmpty( ) )
 {
   front = ( front + 1 ) % QUEUE_SIZE;
   numberOfItems--;
   return queue[( front - 1 ) % QUEUE_SIZE];
 }
 else
   throw new DataStructureException
                ( "Queue empty: cannot dequeue" );
}
```

Description

13.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. You coded the following inside the *main* method of the *Test* class, using the *Player* and *PlayerNode* classes.

```
PlayerNode pn = new PlayerNode( );
Player p = pn.getPlayer( );
p.setID( 10 );            // line 10
```

Description

The code compiles, but at run time you get a *NullPointerException* at line 10.

```
Exception in thread "main" java.lang.NullPointerException:
Cannot invoke "Player.setID(int)" because "<local2>" is null
       at Test.main(Test.java:10)
```

Description

Explain what the problem is and how to fix it.

2. You coded the following in the *main* method of the *Test* class, using the *Player* and *PlayerLinkedList* classes.

```
Player p = new Player( 5,"Ajay","Mario" );
PlayerLinkedList pll = new PlayerLinkedList( );
pll.insert( p );
PlayerNode temp = pll.getHead( );           // line 10
System.out.println( "head is " + temp.toString( ) );
```

Description

At compile time, you get the following error:

```
Test.java:10: error: cannot find symbol
   PlayerNode temp = pll.getHead( ); // line 10
                        ^
   symbol  : method getHead ()
   location: variable pll of type PlayerLinkedList
1 error
```

Description

Explain what the problem is and how to fix it.

3. You coded the following in the *main* method of the *Test class*, using the *Player* and *PlayerLinkedList* classes.

```
Player p = new Player( 5,"Ajay","Mario" );
PlayerLinkedList pll = new PlayerLinkedList( );
pll.insert( p );
if ( pll.delete( 5 ) )      // line 10
      System.out.println( "Successful deletion" );
```

Description

At compile time, you get the following error:

```
Test.java:10: error: incompatible types: Player cannot be
converted to boolean
  if ( pll.delete( 5 ) )   // line 10
                 ^
1 error
```

Description

Explain what the problem is and how to fix it.

4. You coded the following inside the *foo* method of the *PlayerLinkedList* class.

```
PlayerNode current = head;    // line 9
while ( current.getPlayer( ).getID( ) != 99 )
      current = current.getNext( );
// more code here but no problem
```

Description

The code compiles, but when you call *foo* inside *main*, you get a *NullPointerException* at run time at line 10.

```
Exception in thread "main" java.lang.NullPointerException:
Cannot invoke "PlayerNode.getPlayer()" because "<local1>" is null
            at PlayerLinkedList.foo(PlayerLinkedList.java:10)
            at Test.main(Test.java:24)
```

Description

What would be a possible scenario that may have caused this error? Explain how to fix this problem.

13.14.6 Write a Short Program

1. Modify the *PlayerLinkedList* class to include one more method: that method inserts a new player in the third position of the list, *head* being the first position. If the list is empty, the method will insert the new player as the head of the list. Be sure to test your method with the appropriate client code.
2. Modify the *PlayerLinkedList* class to include one more method: that method inserts a new player in the next-to-last position of the list. If the list is empty, the method will insert the new player as the head of the list. Be sure to test your

method with the appropriate client code.

3. Modify the *PlayerLinkedList* class to include one more method: that method inserts a new player in the last position of the list. For this, you cannot use the *tail* instance variable. Be sure to test your method with the appropriate client code.
4. Modify the *PlayerLinkedList* class to include one more method: that method deletes the second node of the list, if there is one. Be sure to test your method with the appropriate client code.
5. Modify the *GenericLinkedList* class to include one more method: that method inserts a new item at a given position (a parameter of the method). If the list is empty, the method will insert the new item as the head of the list. If the value of the parameter is greater than the number of items in the list, then the method inserts at the end of the list. You should consider that *head* is at position 1 in the list. Be sure to test your method with the appropriate client code.
6. Modify the *GenericLinkedList* class to include one more method: that method deletes an item at a given position (a parameter of the method). If the value of the parameter is greater than the number of elements in the list, then no item is deleted and an exception is thrown. Your method should return the item deleted, if any. You should consider that the first node is at position 1 in the list. Be sure to test your method with the appropriate client code.
7. Modify the *PlayerLinkedList* class to include one more method: that method takes a parameter that represents a game. The method inserts a new player at a position just after the first *Player* of the list with a *game* instance variable equal to that game. If there is no such node, then

your method should insert at the end of the list. Be sure to test your method with the appropriate client code.

8. Modify the *PlayerLinkedList* class to include one more method: a traversal that outputs the players in the list until we reach a player with a given *id*; that player's data should not be output. Be sure to test your method with the appropriate client code.
9. Modify the *GenericLinkedList* class to include one more method: a method that returns the *n*th item on the list (*n* is a parameter of the method). If there is no *n*th item on the list, the method should *throw* an exception. Test your method with a client that traverses the list by requesting each item in position order.
10. Modify the *PlayerStackLinkedList* class to include one more method: a method that returns the ID of the last player on the stack. Be sure to test your method with the appropriate client code.
11. Modify the *PlayerQueueLinkedList* class to include one more method: a method that outputs every other player in the queue; that is, it outputs the first player, skips the second, outputs the third player, skips the fourth, and so on. Be sure to test your method with the appropriate client code.
12. Modify the *RecursiveLinkedList* class to include one more method: one that inserts at the end of the list. Be sure to test your method with the appropriate client code.
13. Modify the *RecursiveLinkedList* class to include one more method: one that deletes at the end of the list. Be sure to test your method with the appropriate client code.
14. Modify the *RecursiveLinkedList* class to include one more method: one that deletes at the beginning of the list. Be

sure to test your method with the appropriate client code.

15. Code a class encapsulating a stack of *doubles* using an array of 10 elements. Be sure to test your methods with the appropriate client code.
16. Code a class encapsulating a queue of *chars* using a circular array of 10 elements. Be sure to test your methods with the appropriate client code.

13.14.7 Programming Projects

1. Modify the *PlayerLinkedList* to include two more methods: one that returns the *Player* with the minimum *id*, and one that returns all the games played by players with a given *id*. You also need to include the appropriate client code to test your classes.
2. Modify the *PlayerLinkedList* to include two more methods: one that returns the *Player* with the first name in alphabetical order, and one that returns all of the *ids* of the players playing a given game. You also need to include the appropriate client code to test your classes.
3. Code a class encapsulating a singly linked list of website objects. A website has two attributes: a URL address (a *String*, you do not need to use the existing URL Java class) and 10 or fewer keywords describing the topic of the website. In addition to *insert*, *delete*, *peek*, and *toString*, add one more method: a method that, based on a keyword, returns all URL addresses in the list containing that keyword. Your *delete* method should delete an item based on the value of its URL. You also need to include the appropriate client code to test your classes.
4. Code a class encapsulating a singly linked list of football teams. A football team has three attributes: its nickname,

its number of wins, and its number of losses (assume there are no tied games). In addition to *insert*, *delete*, *peek*, and *toString*, add two more methods: a method that returns the nicknames of the teams with the most wins, and another method that returns the five best teams based on winning percentages (if multiple teams have the same winning percentage, you can return the first five such teams in the list). You also need to include the appropriate client code to test your classes.

5. Code a class encapsulating a singly linked list of HTML tags. We will define a valid HTML tag as a string of characters starting with < and ending with >. In addition to *insert*, *delete*, *peek*, and *toString*, add two more methods: a method that returns *true* or *false*, checking if the list contains valid HTML tags only (as previously defined), and another that counts how many items in the list contain the slash (/) character in them. You also need to include the appropriate client code to test your classes.
6. Code a class encapsulating a singly linked list of stocks. A stock is defined by the following attributes: its ticker symbol (a short word, for instance AMD), its price (for example 54.35), and the company's earnings per share (for example 3.25). In addition to *insert*, *delete*, *peek*, and *toString*, add two more methods: a method that returns the list of all the tickers for the penny stocks (a penny stock is a stock whose price is $1.00 or less), and another method that, given a number representing a price earnings ratio (the price earnings ratio of a stock, also known as P/E ratio, is the price of the stock divided by the earnings per share), returns all the tickers with a price earnings ratio less than or equal to that number. You also need to include the appropriate client code to test your classes.

7. Code a class encapsulating a singly linked list of books. A book is defined by the following attributes: its title, its author, its price, and how many are in stock. In addition to *insert*, *delete*, *peek*, and *toString*, add two more methods: a method that, based on a word, returns all the book titles in the list containing that word, and another returning the list of book titles that are out of stock (i.e., there are quantity 0 in stock). You also need to include the appropriate client code to test your classes.
8. Code a class encapsulating a stack of clothes using an array. A clothing item has the following attributes: its name, its color, and whether it can be washed at high temperature. We will limit our stack to 100 clothing items. In addition to *push*, *pop*, *peek*, and *toString*, add two more methods: a method that returns all the clothing items of a given color, and another method that returns how many clothing items in the stack can be washed at high temperature. You also need to include the appropriate client code to test your classes.
9. Code a class encapsulating a queue of foods using a circular array. A food has the following attributes: its name, the number of calories per serving, and the number of servings per container. We will limit our queue to 100 foods. In addition to *enqueue*, *dequeue*, *peek*, and *toString*, add two more methods: a method that returns the average calories per serving of all the foods in the queue, and another method that returns the food item with the highest “total calories” (i.e., calories per serving times number of servings). You also need to include the appropriate client code to test your classes.
10. Code a class encapsulating a sorted linked list of foods; a

Food class is defined in question 67. Your list should be sorted in ascending order using the name of the food as the key. In addition to *insert*, *delete*, *peek*, and *toString*, add two more methods: a method that returns all the *Food* objects in the list that have a number of calories per serving lower than a given value, and another method that returns all the *Food* objects in the list that are located after a given food name. You also need to include the appropriate client code to test your classes.

11. Look at the documentation of the *LinkedList* class in the *java.util* package. Create a class of your choice and code a client class to test the *LinkedList* class; in particular use the *addFirst*, *add*, and *addLast* methods to build a linked list. Also test the *set* and *get* methods. Do you have access to a *toString* method that returns a *String* representation of the linked list? In this implementation, what is the index of the first element in the list? Do the *get* and *element* methods return a copy of an element of the list or a reference to it?

12. Code a stack class using a generic type; the stack should be represented by an array. You should include *push*, *pop*, and *toString* methods. You also need to include the appropriate client code to test your class. In the client class, you should declare and instantiate stacks using at least two different class types.
13. Code a queue class using generics; the queue should be represented by a circular array. You should include *enqueue*, *dequeue*, and *toString* methods. You also need to include the appropriate client code to test your class. In the client class, you should declare and instantiate queues using at least two different class types.

13.14.8 Technical Writing

1. In this chapter, we coded a linked-list class with just two instance variables: the head node and the number of items in the list. We also said that we did not really need the

number of items in the list. Explain how we can traverse the whole list if the class has only one instance variable, *head*.

2. Consider the *PlayerQueueLinkedList* class presented in this chapter, which includes an instance variable called *tail*, in addition to *head*. We want to make the list circular; that is, *tail* "points to" *head*. If you made the method call *tail.getNext( )*, it would return *head*. Describe why and how you would need to modify the *toString* method of the class (assume you do not know the number of items in the list).

13.14.9 Group Project (for a group of 1, 2, or 3 students)

1. Code a doubly linked, sorted list (in ascending order). Each item of the list will just store an *int*.

   You need to code three classes: *Node*, *SortedList*, and *GroupProject*.

   The Node class has three instance variables, all *private*:

   - an *int*, representing the value stored inside the *Node*
   - a *Node* (*next*)
   - another *Node* (*previous*)

   The methods to code are: constructor (at least one), accessors, mutators.

   The *SortedList* class is a doubly linked list, sorted in ascending order.

   It has two instance variables, both private:

   - an *int*, representing the number of items in the list
   - a *Node*, representing the head node in the list

   The methods to code are:

   - *insert*: this method takes one parameter, an *int*; it has a *void* return value.
   - *delete*: this method takes one parameter, an *int*; it

returns a *boolean* value. If we were successful in deleting the item (i.e., the value of the parameter was found in the list), then we return *true*; if we were not successful, then we want to output a message that the value was not found, and therefore, not deleted, and return *false*.

- *toString*: this method takes no parameters and returns a *String* representation of the list.
- constructor (at least one), and accessors and mutators as appropriate.

All methods should keep the list sorted in ascending order.
The *GroupProject* class contains the *main* method; it should do the following:

- create a *SortedList* object reference
- insert successively the values 25, 17, 12, 21, 78, and 47 in the sorted list
- output the contents of the sorted list using the *toString* method
- delete from the sorted list the value 30, using the *delete* method (obviously, 30 will not be found)
- output the contents of the sorted list using the *toString* method
- delete from the sorted list the value 21, using the *delete* method
- output the contents of the sorted list using the *toString* method

Your *insert* and *delete* methods should work properly in all possible scenarios: inserting in an empty list, inserting at the beginning of a list, inserting in the middle of a list, inserting at the end of a list, deleting from an empty list (cannot delete), deleting an item not in the list (cannot delete), deleting the first item in a list, deleting in the middle of a list, deleting the last item in a list.

© Lois GoBe/Shutterstock.

CHAPTER 14

Running Time Analysis

CHAPTER CONTENTS

Introduction
14.1 Orders of Magnitude and Big-Oh Notation
14.2 Running Time Analysis of Algorithms: Counting Statements
14.3 Running Time Analysis of Algorithms and Impact of Coding: Evaluating Recursive Methods
14.4 **Programming Activity: Tracking How Many Statements Are Executed by a Method**
14.5 Running Time Analysis of Searching and Sorting Algorithms
14.6 Simulation of Various Sorting Algorithms
14.7 Chapter Summary
14.8 Exercises, Problems, and Projects
    14.8.1 Multiple Choice Exercises
    14.8.2 Compute the Running Time of a Method
    14.8.3 Programming Projects
    14.8.4 Technical Writing
    14.8.5 Group Project

Introduction

Today's Internet websites have millions of users. The databases storing data on the web servers have grown in size dramatically to accommodate both the growing number of users and the growing volume of data that is posted by these users.

Scientific applications have also experienced a data explosion. Sensors used in these scientific applications, such as meteorology or fluid mechanics, are becoming more precise while at the same time they are getting cheaper. More and more sensors are being used, and application programs have to manage more and more data.

Programs that handle and manipulate this ever-increasing amount of Big Data need to use algorithms that are well-designed and efficient so that they minimize waiting time for users. Two programs that solve the same problem using different algorithms can result in completely different levels of performance—everything else, in particular the hardware platform, being equal. For example, two search engines performing the same search could run at different speeds: one could return its results in tenths of a second while the other could take several seconds to return results.

Many programmers tend to disregard speed and space (memory utilization) issues when writing code. They rely on increasing hardware performance to solve speed problems and the decreasing cost of memory to solve space problems. However, with the Big Data explosion and the resulting data processing issues that we are experiencing today across many industries, designing efficient algorithms has become more and more important. In this chapter, we will focus on algorithms' speed performance.

When we measure the performance of an algorithm, we use the expression **running time**. We cannot predict a single, precise running time for many algorithms, because the amount of processing depends in large part on the number of inputs and the values of those inputs. So we express the running time of an algorithm as a mathematical function of its inputs. This allows us to compare the relative performance of multiple algorithms. For example, the running time for computing the factorial of an integer varies according to the integer value. Factorials of larger numbers require more processing to compute than factorials of smaller numbers. If we can express the running time of multiple algorithms that compute a factorial as a function of their input, then we can compare the relative efficiency of each algorithm. In other cases, such as sorting an array of integers, the running time depends on the number of array elements. Similarly, if we express the running time as a function of the number of elements in the array, we can compare the relative efficiency of multiple sorting algorithms.

The input value or number of inputs for an algorithm represents the size of the problem for which we are trying to compute the running time. We will call that number n. We are interested in relative time, independent of the hardware platform, not absolute time. Furthermore, we are typically interested in the order of magnitude of the algorithm, rather than a precise mathematical expression as a function of n. Indeed, if n is very large (for example, 1 million or more), performance does not vary noticeably if the algorithm takes n steps or $n + 17$ steps to complete.

However, if an algorithm has a running time expressed as n^2, then the number of inputs has a big impact on performance. For example, we can predict that 10 inputs will require the execution of 100 statements and 1,000 inputs will require the execution of 1 million statements.

The objectives of this chapter are:

- To be able to evaluate the running time of a given algorithm through various methodologies
- To understand that how we code an algorithm directly impacts its running time

of another function

14.1 Orders of Magnitude and Big-Oh Notation

Table 14.1 shows examples of various orders of magnitude for an algorithm as a function of the number of inputs *n*, along with the corresponding number of statement executions for different values of *n*.

Comparisons of Various Functions Representing Running Times

Order of Magnitude	Number of Statements Executed			
	$n = 10$	$n = 20$	$n = 1{,}000$	n = 1 million
$\log n$	2.23	3.23	Approx. 10	Approx. 20
n	10	20	1,000	10^6
$n \log n$	22.3	64.6	Approx. 10,000	Approx. $20*10^6$
n^2	100	400	10^6	10^{12}
n^3	1,000	8,000	10^9	10^{18}
2^n	1,024	Approx. 10^6	Approx. 10^{300}	Approx. 10^{300000}

Description

Let’s look at an example to see how we can use these values. As we will demonstrate later in the chapter, Sequential Search has a running time of *n*, and Binary Search has a running time of *n*. Thus, if we are searching an array of 1 million users for a particular user name, a Sequential Search will take, on average, the execution of an order of 1 million statements, while a Binary Search will require the execution of only 20 statements. Remember, however, that for a Binary Search to work, the array must already be sorted**.** Later in this chapter, we will discuss how to compute these running times.

As we can see from the table, algorithms that have a running time where *n* is the exponent of the function, such as 2, take a very large number of statement executions and are very slow; they should be used only if no better algorithm can be found.

Running times of algorithms are often represented using the **Big-Oh** or the **Big-Theta** notation, as in $O(n)$ or $\Theta(n^2)$, for example. The mathematical definition of Big-Theta is as follows:

A function $f(n)$ is Big-Theta of another function $g(n)$, or $\Theta(g(n))$, if and only if:

1. $f(n)$ is **Big-Omega** of $g(n)$, or $\Omega(g(n))$, i.e., there exist two positive constants, $n1$ and $c1$, such that for any $n >= n1$, $f(n) >= c1 * g(n)$.
   In other words, for n sufficiently big, $g(n)$ is a lower bound of $f(n)$; that is, $g(n)$ is smaller than $f(n)$, if we ignore the constants.
   and
2. $f(n)$ is Big-Oh of $g(n)$, or $O(g(n))$, i.e., there exist two positive constants, $n2$ and $c2$, such that for any $n >= n2$, $f(n) <= c2 * g(n)$.
   In other words, for n sufficiently big, $g(n)$ is an upper bound of $f(n)$; that is, $g(n)$ is bigger than $f(n)$, if we ignore the constants.

It has become common in the industry to say Big-Oh instead of Big-Theta. Indeed, we are really interested in an upper bound running time (Big-Oh), and as tight an upper bound as possible (Big-Theta).

Although the preceding definition may sound a bit complex, when trying to estimate the Big-Oh of a particular function representing a running time, the following rules can be used:

- Keep only the dominant term, i.e., the term that grows the fastest as n grows.
- Ignore the coefficient of the dominant term.

Table 14.2 shows a few examples illustrating these rules.

Examples of Functions Representing Running

Times and Their Respective Big-Oh

f(n)	**Dominant Term**	**Big-Oh**
$2*n+19$	$2*n$	$O(n)$
$3*n^2+6*n+12$	$3*n^2$	$O(n^2)$
$n^3+9*n^2+5*n+2$	n^3	$O(n^3)$
$3*2^n+5*n^3+3*n+7$	$3*2^n$	$O(2^n)$
$n+7*\log n$	n	$O(n)$
$2*n*\log n+8*n+\log n+8$	$2*n*\log n$	$O(n*\log n)$
$3*\log n+35$	$3*\log n$	$O(\log n)$

Description

As an example, we will show that the function f(n) = 3*n2 + 6*n +12 is $\Theta(n^2)$.

First we show that f(n) is $\Omega(n^2)$:

$$\text{For } n >= 0,$$

$$f(n) = 3*n^2 + 6*n + 12 >= 3*n^2$$

Description

So if we choose n1 = 0 and c1 = 3, we just proved by definition that *f(n)* is $\Omega(n^2)$.

Now we show that the same function *f(n)* is $O(n^2)$.

For $n >= 1$, we can rewrite *f(n)* as

$$f(n) = n^2 * (3 + 6/n + 12/n^2)$$

Description

For n >= 6, we have

$$6/n <= 1 \text{ and } 12/n^2 < 1$$

Description

therefore,

$$f(n) <= n^2 * (3+1+1) = 5 * n^2$$

Description

So if we choose n2 = 6 and c2 = 5, we just proved by definition that *f(n)* is $O(n^2)$.

Since *f(n)* is both Big-Omega(n^2) and Big-Oh(n^2), then *f(n)* is Big-Theta(n^2).

To show that a polynomial function is Big-Oh of its most dominant term, we simply factor by the most dominant term as follows:

For $n > 0$,

$$f(n) = a_p\, n^p + a_{p-1}\, n^{p-1} + \ldots + a_2\, n^2 + a_1\, n + a_0, \text{where } a_p \text{ is strictly positive}$$

$$f(n) = a_p\, n^p\, (1 + (a_{p-1} / a_p) 1 / n + \ldots + (a_2 / a_p) 1 / n^{p-2} + (a_1 / a_p) 1 / n^{p-1} + (a_p / a_0) 1 / n^p)$$

$$f(n) <= a_p\, n^p\, (1 + |(a_{p-1} / a_p)| 1 / n + \ldots + |(a_2 / a_p)| 1 / n^{p-2} + |(a_1 / a_p)| 1 / n^{p-1} + |(a_p / a_0)| 1 / n^p)$$

Description

All a_i's are constants; let M be the maximum of all $|(a_i/a_p)|$. Thus,

$$f(n) <= a_p\, n^p\, (1 + M 1 / n + \ldots + M 1 / n^{p-2} + M 1 / n^{p-1} + M 1 / n^p)$$

$$f(n) <= a_p\, n^p\, (1 + M(1 / n + \ldots + 1 / n^{p-2} + 1 / n^{p-1} + 1 / n^p))$$

$$f(n) <= a_p\, n^p\, (1 + M(-1 + 1 + 1 / n + \ldots + 1 / n^{p-2} + 1 / n^{p-1} + 1 / n^p))$$

Description

since we know mathematically that

$1 + a + a^2 + \ldots + a^p = \Sigma a^i$ *from i = 0 to p is equal to* $(1 - a^{p+1}) / (1 - a)$ *for a different from1.*

Using a = 1/*n* and *n* >= 2, we get

$$f(n) <= a_p\, n^p\, (1 + M(-1 + (1 - 1 / n^{p+1}) / (1 - 1 / n)))$$

$$f(n) <= a_p\, n^p\, (1 + M(-1 + (1 - 1 / n^{p+1}) * (n / (n - 1))))$$

Description

Thus

$$f(n) <= a_p\, n^p\, (1 + M\, (-1 + (n\, /\, (n - 1))))$$

$$f(n) <= a_p\, n^p\, (1 + M\, ((-n + 1 + n)\, /\, (n - 1)))$$

$$f(n) <= a_p\, n^p\, (1 + M\, (1\, /\, (n - 1)))$$

Description

Thus

$$f(n) <= a_p\, n^p\, (1 + \mathrm{M}) \text{ for } n >= 2$$

Description

choosing $n_0 = 2$ and $c_0 = a_p\ (1 + M)$.
For n >= n_0, we have

$$f(n) <= c_0\, n^p$$

Description

and therefore,
f(n) is $O(n^p)$, i.e., *f(n)* is Big-Oh of its most dominant term.

14.2 Running Time Analysis of Algorithms: Counting Statements

One simple method to analyze the running time of a code sequence or a method is simply to count the number of times each statement is executed and to calculate a total count of statement executions.

Example 14.1 is a method that calculates the total value of all the elements of an array of size *n* and returns the sum.

Example 14.1 A Single Loop

```
public static int addElements( int [ ] arr )
{
  int sum = 0;                   // ( 1 )
  int i = 0;                     // ( 2 )
    while ( i < arr.length )     // ( 3 )
    {
     sum += arr[i];              // ( 4 )
     i++;                        // ( 5 )
  }
  return sum;                    // ( 6 )
}
```

Description

Let's count how many times each statement is executed.

Assuming the array has *n* elements, we can develop the following analysis:

Statement	**# Times Executed**
(1)	1
(2)	1
(3)	$n + 1$
(4)	n
(5)	n
(6)	1

Description

Note that the loop condition, *i < arr.length*, is executed one more time than each statement of the loop body: when *i* is equal to *arr.length*, we evaluate the loop condition, but we exit the loop and thus do not execute the two statements in the loop body. Thus, the total number of statements executed, *T(n)*, is equal to:

$$
\begin{aligned}
T(n) &= 1 + 1 + (n+1) + n + n + 1 \\
&= 3n + 4 \\
&= O(n)
\end{aligned}
$$

Description

So we can say that the running time of the *addElements* method is *O(n)*. Note that in the end, we do not need an exact count of the statements executed, since we are really interested in the Big-Oh running time of the function.

Example 14.2 is a method that determines the maximum value in a two-dimensional array of *ints*.

Example 14.2 A Double Loop

```
public static int calculateMaximum( int [ ][ ] arr )
{
  int maximum  = arr[0][0];                        // ( 1 )
  for ( int i = 0; i < arr.length; i++ )           // ( 2 )
  {
  for ( int j = 0; j < arr[i].length; j++ )       // ( 3 )
  {
   if ( maximum < arr[i][j] )                     // ( 4 )
        maximum = arr[i][j];                      // ( 5 )
  }
 }
 return maximum;                                  // ( 6 )
```

Description

Let’s count how many times each statement is executed. In order to keep things simple, we assume that the array has n rows and each row has n columns. We can then develop the following analysis:

Statement	**# Times Executed**
(1)	1
(2)	$1 + (n+1) + n = 2* n + 2$
(3)	$n *(1 + (n+1) + n) = 2* n^2 + 2* n$
(4)	$n * n = n^2$
(5)	between 0 and $n * n$
(6)	1

Description

Statement (2) actually contains three statements: *int i = 0* is executed 1 time, *i < arr.length* is executed (n + 1) times as *i* goes from 0 to n, and *i++* is executed n times as *i* is incremented n times.

In evaluating the number of times statements (3), (4), and (5) will be executed, we first note that we will enter the outer loop n times. Statement (3) also contains three statements: *int j = 0* is executed each time we enter the outer loop, or n times; *j < arr.length*[i] is executed n times each time we enter the outer loop, or $n * n$ times, as *j* goes from 0 to n; and *j++* is executed n times each time we enter the outer loop, or $n * n$ times.

Since we enter the outer loop n times and for each outer loop iteration, we enter the inner loop n times, statement (4) will be executed $n * n$ times. As for statement (5), it will be executed once each time the Boolean expression *maximum < arr*[i][j] evaluates to *true*. We cannot tell how many times that will happen, but we can tell that it will happen no more than $n * n$ times. We will call this unknown value x.

Thus, the total number of statements executed, $T(n)$, is equal to:

$$\begin{aligned} T(n) &= 1+(2*n+2)+(2*n^2+2*n)+(n^2)+x+1 \\ &= 3*n^2+4*n+4+x \end{aligned}$$

Description

with $x < n * n$

Furthermore, since the value of x is between 0 and n^2,

$$3 * n^2 + 4 * n + 4 <= T(n) <= 3 * n^2 + 4 * n + 4 + n^2$$

$$3 * n^2 + 4 * n + 4 <= T(n) <= 4 * n^2 + 4 * n + 4$$

Description

since $T(n)$ has both lower and upper bounds that are $O(n^2)$, $T(n)$ is $O(n^2)$.

For our third example, let's compute the running time of a Sequential Search, implemented by the code shown in **Example 14.3**.

Example 14.3 Sequential Search Algorithm

```
public static int sequentialSearch( int [ ] array, int key )
{
  for ( int i = 0; i < array.length; i++ )    // ( 1 )
    if ( array[ i ] == key )                   // ( 2 )
      return i;                                // ( 3 )
  return -1;                                   // ( 4 )
}
```

Description

Let's count how many times each statement is executed. Assuming the array has n elements, we can develop the following analysis:

Statement	**# Times Executed**
(1)	1 + (between 1 and (n + 1)) + (between 0 and n)
(2)	between 1 and n
(3)	0 or 1
(4)	1 or 0

Description

Thus, if $T(n)$ represents the total number of statements executed, we can say that

$$1 + (1) + (0) + 1 + 1 <= T(n) <= 1 + (n + 1) + n + n + 1$$

$$4 <= T(n) <= 3n + 3$$

Description

$T(n) <= 3n + 3$ shows that $T(n)$ is $O(n)$.

However, we cannot really tell, from the coding of the function, how many statements will be executed as a function of *n.* In these situations, it is interesting to consider three running times:

- the worst-case running time
- the best-case running time
- the average-case running time

In the worst case, where the search key is not found in the array or it is found in the last element, $T(n) = 3n+ 3$, and therefore $T(n)$ is $O(n)$, as mentioned earlier.

In the best case, the element we are looking for is at index 0 of the array, and only four statements will be executed, independently of the value of *n*. Thus, the best-case running time is $O(1)$ since we do not take the multiplying constant into consideration when we compute a Big-Oh.

In the average case, we find the element we are looking for in the middle of the array, and the value of $T(n)$ will be

$$\begin{aligned} T(n) &= 1+(n+1)/2+n/2+n/2+1 \\ &= 3n/2+2\tfrac{1}{2} \\ &= O(n) \end{aligned}$$

Description

14.3 Running Time Analysis of Algorithms and Impact of Coding: Evaluating Recursive Methods

In this section, we will learn how to compute the running time of a recursive method. We will also look at how coding a method has a direct impact on its running time.

Consider coding a recursive method that takes one parameter, *n,* and returns 2^n. There are several ways to code that method, and we will consider two of them here so that we can assess which algorithm is more efficient.

Our first method, *powerOf2A*, is designed using this approach:

- when $n = 0$, $2^0 = 1$. This is our base case.
- For our general case, we use this calculation: $2^n = 2 * 2^{n-1}$

This first problem formulation results in the method shown in **Example 14.4**.

Example 14.4 First Recursive Formulation of 2^n

```
public static int powerOf2A( int n ) // n >= 0
{
   if ( n == 0 )
        return 1;
   else
        return 2 * powerOf2A( n - 1 );
}
```

Description

Our second method, *powerOf2B*, is designed using this approach:

- when $n = 0$, $2^n = 1$. This is our base case.
- For our general case, we use this calculation: $2^n = 2^{n-1} +$

2^{n-1}

This second problem formulation results in the method shown in **Example 14.5**.

Example 14.5 Second Recursive Formulation of 2^n

```
public static int powerOf2B( int n ) // n >= 0
{
   if ( n == 0 )
      return 1;
   else
      return powerOf2B( n - 1 ) + powerOf2B( n - 1 );
}
```

Description

Let's compute the running time of *powerOf2A* as a function of the input *n*; we will call it $T1(n)$.

In the base case (*n* is equal to 0), *powerOf2A* makes only one comparison and returns 1. Thus,

$$T1(0) = 1$$

Generally, since it takes $T1(n)$ to compute and return *powerOf2A*(*n*), then it takes $T1\ (n - 1)$ to compute and return *powerOf2A*($n - 1$).

Thus, in the general case, the comparison in the *if* statement will cost us 1 instruction; computing and returning *powerOf2A*(n – 1) will cost us $T1 n - 1$); and multiplying that result by 2 will cost us 1 instruction. Thus, the total time $T1(n)$ can be expressed as follows:

$$\begin{aligned} T1(n) &= 1 + T1(n-1) + 1 \\ &= T1(n-1) + 2 \ // \quad \text{Equation 14.1} \end{aligned}$$

Description

The preceding equation, which we will call Equation 14.1, is called a recurrence relation between $T1(n)$ and $T1 n - 1$)

because $T1(n)$ is expressed as a function of $T1n - 1)$.

From there, we can use a number of techniques to compute the value of $T1(n)$ as a function of n.

Handwaving Method

This method is called handwaving because it is more an estimation method, rather than a method based on strict mathematics.

From the preceding recurrence relation, we can say that it costs us two instructions to go down one step (from n to $n - 1$). Therefore, to go down n steps will cost us $2 * n$ instructions. We then add one instruction for $T(0)$, and get

$$T1(n) = 2 * n + 1$$

Iterative Method

This method involves iterating several times, starting with the recurrence relation until we can identify a pattern. In general, we can say that

$$T1(x) = T1(x-1) + 2, \text{where } x \text{ is some integer} \quad // \text{ Equation 14.2}$$

Description

We call this Equation 14.2, which is the same as Equation 14.1, except that x has been substituted for n.

We now want to express $T(n)$ as a function of $T(n - 2)$; thus, we want to replace $T(n - 1)$ in Equation 14.1 by an expression using $T(n - 2)$.

Substituting $n - 1$ for x in Equation 14.2, we get

$$T1(n-1) = T1(n-2) + 2$$

Description

Plugging the value of $T1(n - 1)$ into Equation 14.1, we get

$$\begin{aligned} T1(n) &= T1(n-2)+2+2 \\ &= T1(n-2)+2*2 \quad // \text{ Equation 14.3} \end{aligned}$$

Description

Note that in Equation 14.3, we do not simplify 2 ∗ 2. In this way, we are trying to let a pattern develop so we can easily identify it.

SOFTWARE ENGINEERING TIP

When trying to develop and identify a pattern using iteration, do not precisely compute all the terms. Instead, leave them as patterns.

Using $x = n - 2$ in Equation 14.2, we get

$$T1(n-2) = T1(n-3)+2$$

Description

Plugging the value of $T1(n-2)$ into Equation 14.3, we get

$$\begin{aligned} T1(n) &= T1(n-3)+2+2*2 \\ &= T1(n-3)+2*3 \quad // \text{ Equation 14.4} \end{aligned}$$

Description

Using $x = n - 3$ in Equation 14.2, we get

$$T1(n-3) = T1(n-4)+2$$

Description

Plugging the value of $T1(n-3)$ into Equation 14.4, we get

$$\begin{aligned} T1(n) &= T1(n-4)+2+2*3 \\ &= T1(n-4)+2*4 \end{aligned}$$

Description

Now we can see the pattern as follows:

$$T1(n) = T1(n-k) + 2 * k, \text{where } k \text{ is an integer between 1 and } n \quad // \text{ Equation 14.5}$$

Description

Plugging $k = n$ in Equation 14.5 in order to reach the base case of *T*10), we get

$$T1(n) = T1(0) + 2 * n = 1 + 2 * n = 2 * n + 1$$

Description

Proof by Induction Method

If we can guess the value of *T*1(*n*) as a function of *n,* then we can use a proof by induction in order to prove that our guess is correct. We can use the preceding iteration method to come up with a guess for *T*1(*n*).

Generally, a proof by induction works as follows:

- Verify that our statement (equation in this case) is true for a base case.
- Assume that out statement is true up to *n.*
- Prove that it is true for *n* + 1.

Let's go through the induction steps with our guess that *T*1(*n*) = 2 ∗ *n* + 1, which we may have generated from our iterative or handwaving method.

Step 1: Verify that the value that our guess gives to *T*1(0) is correct.

$$\begin{aligned} T1(0) &= 2 * 0 + 1 \\ &= 1 \end{aligned}$$

Description

Thus, our guess is correct for $T1(0)$.
Step 2: Assume that $T1(n) = 2 * n + 1$.
Step 3: Prove that $T1(n + 1) = 2 *(n + 1) + 1$.
Plugging $x = n + 1$ in Equation 14.2, we get

$$T1(n+1) = T1(n) + 2$$

Description

Then, using our assumption and replacing $T1(n)$ by $2 * n + 1$, we get

$$\begin{aligned} T1(n+1) &= 2 * n + 1 + 2 \\ &= 2 * n + 2 + 1 \\ &= 2 * (n+1) + 1 \end{aligned}$$

Description

Thus, we just proved, by induction, that our guess $T1(n) = 2 * n + 1$ is correct.

Other Methods

When applicable, another method is to use the Master Theorem, but that is beyond the scope of this text.

So the running time of *powerOf2A*(n) is $2 * n + 1$, or $O(n)$.

Let's now compute the running time of *powerOf2B* as a function of the input n. We will call it $T2(n)$.

In the base case (n is equal to 0), *powerOf2B* takes only one comparison to return 1. Thus,

$$T2(0) = 1$$

Generally, since it takes *T*2(*n*) to compute and return *powerOf2B*(*n*), then it takes *T*2(*n* – 1) to compute and return *powerOf2B*(*n* – 1). Thus, in the general case, the comparison in the *if* statement will cost us one instruction; computing and returning *powerOf2B*(*n* – 1) will cost us *T*2 (*n* – 1); doing it a second time will cost us another *T*2(*n* – 1); and adding the two and returning the sum as the result will cost us one instruction. Thus, the total time *T*2(*n*) can be expressed as follows:

$$\begin{aligned} T2(n) &= 1+T2(n-1)+T2(n-1)+1 \\ &= 2*T2(n-1)+2 \text{ // Equation 14.6} \end{aligned}$$

Description

From there, we will use the iteration method in order to compute the value of *T*2(*n*) as a function of *n*.

Substituting *x* for *n*, we can rewrite Equation 14.6 as follows:

$$T2(x)=2*T2(x-1)+2 \text{ // Equation 14.7}$$

Description

Using $x = n - 1$ in Equation 14.7, we get

$$T2(n-1)=2*T2(n-2)+2$$

Description

Plugging the value of *T*2(*n* – 1) into Equation 14.6, we get

$$\begin{aligned} T2(n) &= 2*(2*T2(n-2)+2)+2 \\ &= 2^2*T2(n-2)+2^2+2 \text{ // Equation 14.8} \end{aligned}$$

Description

Again, we leave $2^2 + 2$ as an expression to try to let a pattern develop.

Using $x = n - 2$ in Equation 14.7, we get

$$T2(n-2) = 2 * T2(n-3) + 2$$

Description

Plugging the value of *T*2(*n* – 2) into Equation 14.8, we get

$$\begin{aligned} T2(n) &= 2^2 * (2 * T2(n-3) + 2) + 2^2 + 2 \\ &= 2^3 * T2(n-3) + 2^3 + 2^2 + 2 \quad \text{// Equation 14.9} \end{aligned}$$

Description

Using *x* = *n* – 3 in Equation 14.7, we get

$$T2(n-3) = 2 * T2(n-4) + 2$$

Description

Plugging the value of *T*2(*n* – 3) into Equation 14.9, we get

$$\begin{aligned} T2(n) &= 2^3 * (2 * T2(n-4) + 2) + 2^3 + 2^2 + 2 \\ &= 2^4 * T2(n-4) + 2^4 + 2^3 + 2^2 + 2 \quad \text{// Equation 14.10} \end{aligned}$$

Description

Now we can see the pattern as follows:

$$T2(n) = 2^k * T2(n-k) + 2^k + 2^{k-1} + \ldots + 2^2 + 2,$$
$$\text{where } k \text{ is an integer between 1 and } n \quad \text{// Equation 14.11}$$

Description

Noting that

$$\begin{aligned} 2^k + 2^{k-1} + \ldots + 2^2 + 2 &= -1 + 2^k + 2^{k-1} + \ldots + 2^2 + 2 + 1 \\ &= -1 + (2^{k+1} - 1) / (2 - 1) \\ &= 2^{k+1} - 2 \end{aligned}$$

Description

Equation 14.11 becomes

$$T2(n) = 2^k * T2(n-k) + 2^{k+1} - 2,$$

where k is an integer between 1 and n // Equation 14.12

Description

Plugging $k = n$ in Equation 14.12 in order to reach the base case of $T2(0)$, we get

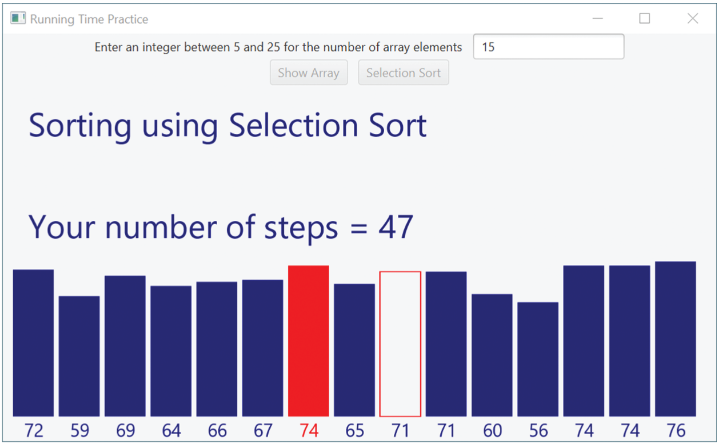
$$\begin{aligned} T2(n) &= 2^n * T2(0) + 2^{n+1} - 2 \\ &= 2^n * 1 + 2^{n+1} - 2 \\ &= 2^n + 2^{n+1} - 2 \\ &= 2^n (1 + 2) - 2 \\ &= 3 * 2^n - 2 \\ &= O(2^n) \end{aligned}$$

Description

Thus, *powerOf2A* runs in $O(n)$ while *powerOf2B* runs in $O(2^n)$, although they perform the same function.

As a result, computing 2^{20} using *powerOf2A* will cost 20 statement executions, while computing 2^{20} using *powerOf2B* will cost 1 million statement executions.

This simple example shows that how we code a method can have a significant impact on its running time.


Running Time Practice
Enter an integer between 5 and 25 for the number of array elements
15
Show Array
Selection Sort

Sorting using Selection Sort

Your number of steps = 47

72 59 69 64 66 67 74 65 71 71 60 56 74 74 76

14.4 Programming Activity: Tracking How Many Statements Are Executed by a Method

In this activity, you will work with a variable-size integer array. Specifically, you will perform the following operations:

1. Write code to keep track of the number of statement executions during a selection sort.
2. Run a simulation to compute the number of statements executed as a function of the number of elements in the array.
3. Estimate the running time of Selection Sort as a function of *n*, the number of elements in the array being sorted.

The framework for this Programming Activity will animate your algorithm so that you can perform a simulation on the number of statement executions inside the *selectionSort* method compared to the number of elements in the array that is sorted. For example, **Figure 14.1** shows the current number of statement executions for an array of 15 elements.

Figure 14.1 Animation of the Programming Activity

Description

At this point, the application has executed 47 statements.

Instructions

In this chapter's Programming Activity folder, you will find the source files needed to complete this activity. Copy all the files to a folder on your computer. Note that all files should be in the same folder.

Open the *RunningTimePracticeController.java* source file. Searching for five asterisks (*****) in the source code will position you at the sample method where you will add your code. In this task, you will fill in the code for the *selectionSort* method in order to keep track of the number of statement executions needed to sort an array using the Selection Sort algorithm. You should not instantiate the array; we have done that for you. **Example 14.6** shows the section of the *RunningTimePracticeController* source code where you will add your code.

Example 14.6 Location of Student Code in *RunningTimePracticeController*

```
//  1. *****  student writes this method
/**  Sorts arr in ascending order using the selection sort algorithm
*    Adds a counter to count the number of statement executions
*/
public int selectionSort( )
{
 // Note:  To count the number of statement executions, use a counter
 // The variable counter has been declared and initialized for you
 // at the beginning of this method
 // Inside the body of the inner loop, increment the counter
 // Replace the return statement so that this method returns the value of
 // the counter. To slow down or accelerate the animation, modify the
 // argument of Pause.wait in the handle method of the ArrayAnimationTimer inner class
 int counter = 0;
 int temp, indexOfMax;
 for ( int i = 0; i < arr.length - 1; i++ )
 {
    // find index of largest value in the subarray
    indexOfMax = 0;
    animate( i,  0,  0,  counter );
    for ( int j = 1; j < arr.length - i; j++ )
    {
      if ( arr[j] > arr[indexOfMax] )
        indexOfMax = j;
      animate( i, j, indexOfMax, counter );
    }
     // swap arr[indexOfMax] and arr[arr.length - i - 1]
    temp = arr[indexOfMax];
    arr[indexOfMax] = arr[arr.length - i - 1];
    arr[arr.length - i - 1] = temp;
 }
 animate( arr.length  -  1,  0,  0,  counter );
 return 0;
} // end of selectionSort
```

Description

Note that we provide a dummy *return* statement (*return* 0;). We do this so that the source code will compile. Just replace the dummy *return* statement with the appropriate *return* statement for the method.

Our framework will animate your algorithm so that you can watch your code work. If you want to accelerate or slow down the animation, modify the argument of *Pause.wait* in the *handle* method of the *ArrayAnimationTimer* inner class.

To test your code, compile *RunningTimeController.java* and run the *RunningTimePracticeApplication*. When the program begins, you will see a text box. Enter the number of elements for the array, and press the “Show Array” button to populate the array. Then press the “Selection Sort” button to execute your selection sort code. Because the values of the array are randomly generated, the values will be different each time the program runs.

Troubleshooting

If the animation is incorrect, and you think your method does return a correct value for the counter, verify that you correctly incremented the counter inside the inner loop.

In order to derive a closed-end expression for the number of statement executions as a function of the size of the array, follow these tips:

- If n is the size of the array, compare n, n^2, n^3, n^4, …, 2^n, to the value of the counter.
- When doing the preceding, divide n, n^2, n^3, n^4, …, 2^n by the number of statements executed.

DISCUSSION QUESTIONS

1. What is the value of the counter with the following array sizes: 5, 10, 15, 20, 25?
2. In relation to n, the size of the array, what is the value of the counter?
3. What is the running time of Selection Sort in Big-Oh notation?
4. If the array is already sorted in either the correct or opposite order, does that make a difference in the number of statement executions? What can you say about the worst-case and best-case running times?

14.5 Running Time Analysis of Searching and Sorting Algorithms

In studying the running time of various searching and sorting algorithms, we will look at the following scenarios:

- best case
- worst case
- average case

Some methods have a very efficient running time. We mentioned earlier that the running time of Binary Search was log *n*. Thus, searching a sorted array of 1 billion items using Binary Search will take only 30 statement executions since log (1 billion) is approximately 30.

Example 14.7 shows the code for a recursive binary search.

Example 14.7 Recursive Binary Search

```
public static int recursiveBinarySearch
                 ( int [ ] arr, int key, int start, int end )
{
  if ( start <= end )
  {
   // look at the middle element of the subarray
   int middle = ( start + end ) / 2;
   if ( arr[middle] == key )      // found key, base case
     return middle;
   else if ( arr[middle] > key )  // look lower
     return recursiveBinarySearch( arr, key, start, middle - 1 );
   else                           // look higher
     return recursiveBinarySearch( arr, key, middle + 1, end );
  }
   else                           // key not found, base case
     return -1;
}
```

Description

In the best-case scenario, we will find the search value exactly in the middle of the array, at the array index we check first. Thus, the best-case running time of Binary Search is *O*(1). In the worst-case scenario, we will not find the search value in the array. **Table 14.3** shows the number of elements in the subarray we are searching versus the comparison count for an initial array of 1024 elements. To keep things simple, we halved the size of the subarray after each comparison (in reality, the element in the middle is eliminated). The total of *11* comparisons for an array of *1024* elements suggests a running time of *log n*. Let's compute the running time of the worst-case scenario.

Worst Case Number of Comparisons for Binary Search

Number of Elements in Subarray As We Are Searching	Comparison Count
1024	1
512	2
256	3
128	4
64	5
32	6

16	7
8	8
4	9
2	10
1	11

Description

In the general case, the comparison of the first *if* statement will cost us one instruction; the assignment statement will cost us two instructions; the comparison in the second *if* statement will cost us one instruction; the comparison in the *else/if* statement will also cost us one instruction; computing and returning *recursiveBinarySearch*(*arr, key, start, middle* – 1) or *recursiveBinarySearch*(*arr, key, middle* + 1, *end*) will cost us $T(n/2 - 1)$ or $T(n/2)$ instructions. Note that only one recursive call will be made. Thus, the total time $T(n)$ can be expressed as follows:

$$\begin{aligned} T(n) &= 1+2+1+1+T(n/2) \\ &= T(n/2)+5 \text{ // Equation 14.13} \end{aligned}$$

Description

In the base case (n is equal to 1), *recursiveBinarySearch* makes only the first comparison, one addition, one division, the second comparison, and then returns the index of the found element or –1. Thus,

$$T(1) = 5.$$

From there, we will use the iteration method in order to compute the value of $T2(n)$ as a function of n.

Substituting x for n, we can rewrite Equation 14.13 as follows:

$$T(x) = T(x/2)+5 \text{ // Equation 14.14}$$

Using $x = n/2$ in Equation 14.14, we get

$$\begin{aligned} T(n/2) &= T((n/2)/2)+5 \\ &= T(n/2^2)+5 \end{aligned}$$

Description

Plugging the value of $T(n/2)$ into Equation 14.13, we get

$$\begin{aligned} T(n) &= (T(n/2^2)+5)+5 \\ &= T(n/2^2)+5*2 \quad \text{// Equation 14.15} \end{aligned}$$

Description

Using $x = n/2^2$ in Equation 14.14, we get

$$\begin{aligned} T(n/2^2) &= T((n/2^2)/2)+5 \\ &= T(n/2^3)+5 \end{aligned}$$

Description

Plugging the value of $T(n/2^2)$ into Equation 14.15, we get

$$\begin{aligned} T(n) &= (T(n/2^3)+5)+5*2 \\ &= T(n/2^3)+5*3 \quad \text{// Equation 14.16} \end{aligned}$$

Description

Using $x = n/2^3$ in Equation 14.14, we get

$$\begin{aligned} T(n/2^3) &= T((n/2^3)/2)+5 \\ &= T(n/2^4)+5 \end{aligned}$$

Description

Plugging the value of $T(n / 2^3)$ into Equation 14.16, we get

$$\begin{aligned} T(n) &= (T(n / 2^4) + 5) + 5 * 3 \\ &= T(n / 2^4) + 5 * 4 \quad \text{// Equation 14.17} \end{aligned}$$

Description

Now we can see the pattern as follows:

$$T(n) = T(n / 2^k) + 5 * k,$$

Description

We now want to choose k such that $n / 2^k$ is equal to 1 in order to reach our base case. If $n / 2^k = 1$, then $n = 2^k$, and taking the log of each side:

$$\begin{aligned} \log n &= \log 2^k \\ &= k \log 2 \\ &= k * 1 \\ &= k \end{aligned}$$

Description

Plugging $k = \log n$ in Equation 14.18, we get

$$
\begin{aligned}
T(n) &= T(1) + 5 * \log n \\
&= 2 + 5 * \log n \\
&= O(\log n)
\end{aligned}
$$

Description

Thus, Binary Search is $O(\log n)$ in the worst case. Note that the value of the original constant, here 5, does not impact the order of magnitude of the running time.

In the average case, we will find the search value after performing half the number of comparisons as in the worst-case scenario. Thus, the average running time of binary search is also $O(\log n)$.

Now, let’s calculate the running time of Insertion Sort as a function of n, the number of elements in the array. The code for an Insertion Sort is shown in **Example 14.8**.

Example 14.8 Insertion Sort

```
/**  Performs an Insertion Sort on an integer array
*    @param array   array to sort
*/
public static void insertionSort( int [  ] array )
{
  int j, temp;
  for ( int i = 1; i < array.length; i++ )
  {
      j = i;
      temp = array[i];
      while ( j != 0 && array[j - 1] > temp )
      {
        array[j] = array[j - 1];
        j--;
      }
      array[j] = temp;
  }
}
```

Description

The *for* loop header will execute n times. We will execute the body of the *for* loop $n - 1$ times.

In the best case, the array is already sorted. In this case, the *while* loop condition will always evaluate to *false*, and we will never execute the *while* loop body. So inside the *for* loop, the three statements and the loop condition will each execute once for each iteration of the *for* loop, thus executing a total of $4 * (n - 1)$ times. Therefore, the best-case running time is $O(n)$.

In the worst case, the array is sorted in the opposite order. In this case, the *while* loop condition will always be *true* for its first evaluation, and we will enter the *while* loop every time we iterate the *for* loop. Thus, the two statements inside the *while* loop will each execute $(1 + 2 + 3 + 4 + \ldots + (n - 1))$ times. Since $(1 + 2 + 3 + 4 + \ldots + (n - 1)) = n * (n - 1) / 2$, the worst-case running time of insertion sort is $O(n^2)$.

In the average case, we will enter the *while* loop half the times we try. The average case is still $O(n^2)$.

Bubble Sort, like Insertion Sort, is implemented with a double loop and runs also $O(n^2)$.

Merge Sort and Quick Sort are two sorting algorithms implemented recursively.

The pseudocode for Merge Sort is as follows:

- If the array has only one element, it is already sorted, thus do nothing; otherwise:
    - Merge sort the left half of the array.
    - Merge sort the right half of the array.
    - Merge the two sorted half arrays into one so that the resulting array is sorted.

The last operation involves looping through all the elements of the two half-arrays; it takes $O(n)$; thus, we can derive the following recursive formulation for its running time of Merge Sort:

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + n \\ &= 2T(n/2) + n \end{aligned}$$

Description

Using derivation, we get

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/2^2) + n/2) + n \\ &= 2^2 T(n/2^2) + 2n \end{aligned}$$

Description

Continuing to iterate,

$$\begin{aligned} T(n) &= 2^2\, T(n/2^2) + 2n \\ &= 2^2\,(2\,T(n/2^3) + n/2^2) + 2n \\ &= 2^3\, T(n/2^3) + 3n \end{aligned}$$

$$\begin{aligned} T(n) &= 2^3\, T(n/2^3) + 3n \\ &= 2^3\,(2\,T(n/2^3) + n/2^3) + 3n \\ &= 2^4\, T(n/2^4) + 4n \end{aligned}$$

Description

Thus, we identify the general pattern

$$T(n) = 2^k\, T(n/2^k) + kn$$

Description

Choosing k so that $n/2^k = 1$ in order to reach the base case, i.e., $n = 2^k$, $k = \log n$, we get

$$\begin{aligned} T(n) &= n\,T(1) + n\log n \\ &= O(n\log n) \end{aligned}$$

Description

So Merge Sort is $O(n \log n)$, which is better than Insertion Sort, Bubble Sort, and Selection Sort. It is the same for best-case, worst-case, and average-case scenarios.

The running time of Quick Sort is also $O(n \log n)$ in the average-case scenario but is $O(n^2)$ in the worst-case scenario. The worst-case scenario for Quick Sort happens when the array is already sorted.

CODE IN ACTION

Within the online resources, you will find a movie with a step-by-step illustration of how to compute running times for various methods. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock

with these end-of-chapter questions

14.8.1	Multiple Choice Exercises Questions 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
14.8.2	Compute the Running Time of a Method Questions 11, 12, 13, 14, 15, 16, 17, 18
14.8.4	Technical Writing Question 27

```
11 379 418 452 531 710 1118 1169 1177 ...
Unsorted array: Insertion sort count = 259873
11 379 418 452 531 710 1118 1169 1177 ...
Sorted array: Insertion sort count = 1023
43 471 589 643 653 845 1050 1065 1089 ...
Unsorted array: Merge sort count = 8917
43 471 589 643 653 845 1050 1065 1089 ...
Sorted array: Merge sort count = 5120
31 38 77 192 222 252 547 560 671 673 ...
Unsorted array: Quick sort count = 10332
31 38 77 192 222 252 547 560 671 673 ...
Sorted array: Quick sort count = 523776
```

14.6 Simulation of Various Sorting Algorithms

A way to validate a running time is to run some simulation and count the statements, in particular the comparison statements, that execute as the program runs. **Example 14.9** is a simulation program that counts the number of comparisons for insertion sort, merge sort, and quick sort. We declare and initialize count, a *static* variable, at line 10. Before we run each sorting algorithm, we reinitialize count to *0* (lines 23, 31, 38, 46, 53). Inside each sorting method, we increment *count* each time we make a comparison (lines 100, 157, 229). We call each sorting method twice, first on an unsorted array, and then on the sorted array. **Figure 14.2** shows the output. **Table 14.4** compares the simulation results to mathematical expressions as a function of *n*, the number of elements in the array. In this example, we use a value of *1024* for *n* (line 9), but it can easily be changed to make further simulation runs. Our simulation confirms our earlier math analysis: insertion sort runs in $O(n^2)$ and merge sort and quick sort run in $O(n \log n)$.

Example 14.9 Simulation of Various Sorting Algorithms

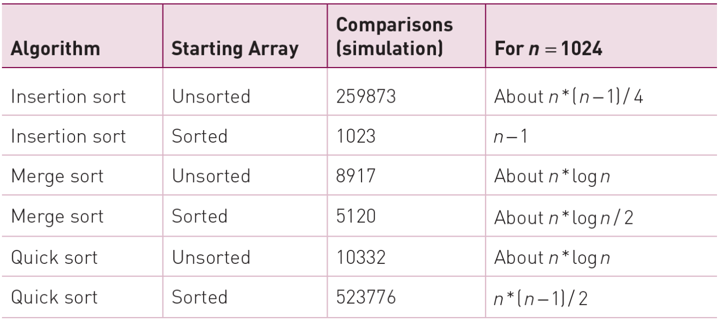
```java
/* Simulation of various sorting algorithms
   Anderson, Franceschi
*/

import java.util.*;

public class Simulation
{
  public static int SIZE = 1024;
  public static int count = 0;

  public static void main( String [] args )
  {
     int [] grades = generateArray( SIZE );

     // sort unsorted array using insertion sort
     insertionSort( grades );
     output( grades );
     System.out.println( "Unsorted array: " +
        "Insertion sort count = " + count );

     // sort sorted array using insertion sort
     count = 0;
     insertionSort( grades );
     output( grades );
     System.out.println( "Sorted array: " +
        "Insertion sort count = " + count );

     grades = generateArray( SIZE );
     // sort unsorted array using merge sort
     count = 0;
     mergeSort( grades, 0, grades.length - 1 );
     output( grades );
     System.out.println( "Unsorted array: " +
        "Merge sort count = " + count );

     // sort sorted array using merge sort
     count = 0;
     mergeSort( grades, 0, grades.length - 1 );
     output( grades );
     System.out.println( "Sorted array: " +
        "Merge sort count = " + count );

     grades = generateArray( SIZE );
     // sort unsorted array using quick sort
     count = 0;
     quickSort( grades, 0, grades.length - 1 );
     output( grades );
     System.out.println( "Unsorted array: " +
        "Quick sort count = " + count );

     // sort sorted array using quick sort
     count = 0;
     quickSort( grades, 0, grades.length - 1 );
     output( grades );
     System.out.println( "Sorted array: " +
        "Quick sort count = " + count );
  }

  /**
  * output method: outputs arr
  * @param arr, an array of ints
  */
  public static void output( int [] arr )
  {
    for ( int i = 0; i < arr.length - 1; i++ )
      System.out.print( arr[i]  + " " );
    System.out.println( );
  }

  /**
  * generateArray method: generates an array of random ints
  * @param n, an int greater than 0
  * @return an array of n random integers
  */
  public static int [] generateArray( int n )
  {
    int [] arr = new int[n];
    Random rand = new Random( );
    int number  = 0;
    for( int i = 0; i < n; i++ )
    {
      number = rand.nextInt( 100000 );
      arr[i] = number;
    }
    return arr;
  }

  /**
  * insertionSort method: sorts arr in ascending order
  * @param arr, an array of ints
  */
  public static void insertionSort( int [] arr )
  {
    int j, temp;
    for ( int i = 1; i < arr.length; i++ )
    {
      j = i;
      temp = arr[i];
      count++;
      while ( j != 0 && arr[j - 1] > temp )
      {
        count++;
        arr[j] = arr[j - 1];
        j--;
      }
      arr[j] = temp;
    }
  }

 /**
  * mergeSort method: sorts the subarray of arr
  * (from index start to index end, both included)
  * in ascending order
  * @param arr, an array of ints
  * @param start an int, the starting index of the subarray
  * @param end an int, the ending index of the subarray
  */
  public static void mergeSort( int [] arr, int start, int end )
  {
    if( start < end ) // general case
    {
      int middle = ( start + end ) / 2;
      // sort the left part of the array
      mergeSort( arr, start, middle );
      // sort the right part of the array
      mergeSort( arr, middle + 1, end );
      // merge the 2 parts
      merge( arr, start, middle, middle + 1, end );
    }
    // else, base case, empty or 1 element array, already sorted
  }

 /**
  * merge method: merges, in ascending order,
  * the two subarrays of arr defined as follows:
  * left subarray: between indexes start1 and end1, both included
  * right subarray: between indexes start2 and end2, both included
  * @param arr, an array of ints
  * @param start1 an int, the starting index of the left subarray
  * @param end1 an int, the ending index of the left subarray
  * @param start2 an int, the starting index of the right subarray
  * @param end2 an int, the ending index of the right subarray
  */
  public static void merge( int [] arr, int start1, int end1,
                                        int start2, int end2 )
  {
    // merge the 2 sides into a temp array
    int [] temp = new int[arr.length]; // temporary array
    int i = start1;
    int j = start2;
    int k = start1;

    // generate merged subarray until one side is done
    while( i <= end1 && j <= end2 )
    {
      count++;
      if( arr[i] < arr[j] )
      {
        temp[k] = arr[i];
        i++;
      }
      else
      {
        temp[k] = arr[j];
        j++;
      }
      k++;
    }

    // flush left subarray if necessary
    while( i <= end1 )
    {
      temp[k] = arr[i];
      i++;
      k++;
    }

    // flush right subarray if necessary
    while( j <= end2 )
    {
      temp[k] = arr[j];
      j++;
      k++;
    }

    // write back temp into arr
    for( i = start1; i <= end2; i++ )
    {
      arr[i] = temp[i];
    }
  }

   /**
  * quickSort method: sorts the subarray of arr
  * (from index start to index end, both included)
  * in ascending order
  * @param arr, an array of ints
  * @param start an int, the starting index of the subarray
  * @param end an int, the ending index of the subarray
  */
  public static void quickSort( int [] arr, int start, int end )
  {
    if ( end > start ) // general case, >= 2 elements
    {
      int pivotIndex = partition( arr, start, end );
      // quick sort left subarray
      quickSort( arr, start, pivotIndex - 1 );
      // quick sort right subarray
      quickSort( arr, pivotIndex + 1, end );
    }
    // else, 0 or 1 element ==> subarray is sorted, do nothing
  }

  /**
  * partition method: partitions the subarray of arr
  * (from index start to index end, both included)
  * around the pivot, arr[start]
  * @param arr, an array of ints
  * @param start an int, the starting index of the subarray
  * @param end an int, the ending index of the subarray
  */
  public static int partition( int [] arr, int start, int end )
  {
    int j = start;
    int pivot = arr[start];
    for ( int i = start + 1; i <= end; i++ )
    {
      count++;
      if ( arr[i] < pivot )
      {
        j++;
        swap( arr, i, j );
      }
    }
    swap( arr, start, j );
    return j;
  }

  /**
  * swap method: swaps the elements of arr
  * at index i and index j
  * @param arr, an array of ints
  * @param i an int, a valid index of arr
  * @param j an int, a valid index of arr
  */
  public static void swap( int [] arr, int i, int j )
  {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }
}
```

Description

Figure 14.2 Output of Example 14.9

Description

Sorting Simulation Counts for an Array of 1024 Elements



Algorithm	Starting Array	Comparisons (simulation)	For $n = 1024$
Insertion sort	Unsorted	259873	About $n*(n-1)/4$
Insertion sort	Sorted	1023	$n-1$
Merge sort	Unsorted	8917	About $n*\log n$
Merge sort	Sorted	5120	About $n*\log n/2$
Quick sort	Unsorted	10332	About $n*\log n$
Quick sort	Sorted	523776	$n*(n-1)/2$

Description

It also shows that insertion sort runs in $O(n)$ if the array is already sorted. This is due to the fact that each insertion takes $O(1)$ and there are $n - 1$ insertions when the array is already sorted. Quick sort runs in $O(n^2)$ when the array is already sorted, which is worse than its running time of $O(n \log \mathbf{n})$ when the array is unsorted. Indeed, when the array is already sorted, *partition* splits the array into a subarray of *1* element and a subarray of $n - 1$ elements at each call of *partition*, the worst possible scenario, dividing the array as unevenly as possible. Merge sort shows a very slight improvement, but not significant in terms of Big-Oh, in the number of comparisons when the array is already sorted: We make exactly $n / 2$ comparisons in the merge step because the elements in the left half of the array are all less than the elements in the right half of the array.

© Lois GoBe/Shutterstock.

CHAPTER REVIEW

14.7 Chapter Summary

- The running time of an algorithm is expressed as a function of its inputs or its number of inputs.
- Orders of magnitude are, in increasing order of execution time: constant, log, polynomial, and exponential. Exponential running times are undesirable.
- Big-Oh notation is the industry standard notation for running times.
- Considering a mathematical function that represents a running time of an algorithm, that function is Big-Oh of its most dominant term, and we ignore the coefficient of that term.
- The coding of a method directly impacts its running time.

14.8 Exercises, Problems, and Projects

14.8.1 Multiple Choice Exercises

1. What is the Big-Oh of this function:
   $T(n)=n^2 - 2n + 99$

   - $O(n^2)$
   - $O(99)$
   - $O(n)$
   - $O(1)$

6. What is the Big-Oh of this function:
   $T(n) = n^3 + 10n^2 + 20n + 30$

   - $O(n^3)$
   - $O(n^2)$
   - $O(n)$
   - $O(1)$

11. What is the Big-Oh of this function:
    $T(n) = n^2 + n*\log n + 12n + 5$

    - $O(n * \log n)$
    - $O(n^2)$
    - O$O(n)$
    - $O(1)$

16. We have the following recurrence relation representing the running time of a function; what is the running time of that function?
    $T(n) = T(n - 1) + 1$

    - $O(2_n)$

- $O(n * \log n)$
- $O(n^2)$
- $O(n)$

21. Which of these running times is the worst?

- $O(n^5)$
- $O(2^n)$
- $O(n * \log n)$
- $O(n)$

26. Look at the following method:

```
public static int foo1( int n )
{
  if ( n > 1 )
   return ( 2 * foo1( n / 4 ) );
  else
   return 1;
}
```

Description

What recurrence formulation best illustrates the running time of the preceding method?

- $T(n) = T(n * 4) + 3$
- $T(n) = T(n / 4) + 3$
- $T(n) = T(n - 4) + 3$
- $T(n) = T(n + 4) + 3$

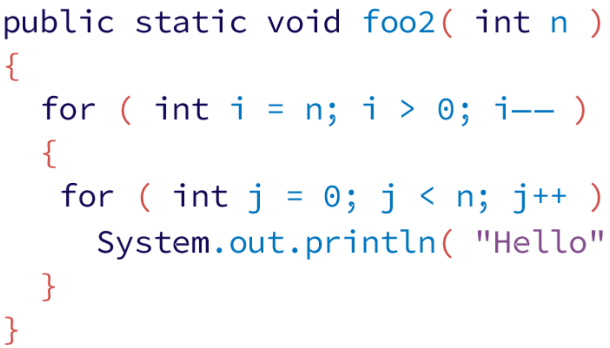
31. What is $\Sigma\, i$ for $i = 1$ to n equal to?

- n^2
- $n * (n + 1) / 2$
- $2n$
- n

36. What is Σ 1 for $i = 1$ to n equal to?

- n^2
- n
- $n * (n + 1) / 2$
- i

41. What is the running time of the *foo2* method?

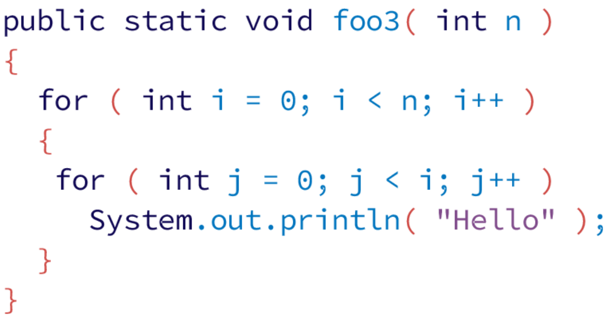


```
public static void foo2( int n )
{
  for ( int i = n; i > 0; i-- )
  {
   for ( int j = 0; j < n; j++ )
     System.out.println( "Hello"
  }
}
```

Description

- $O(n^4)$
- $O(n^3)$
- $O(n^2)$
- $O(n)$

46. What is the running time of the *foo3* method?



```
public static void foo3( int n )
{
  for ( int i = 0; i < n; i++ )
  {
   for ( int j = 0; j < i; j++ )
     System.out.println( "Hello" );
  }
}
```

Description

- $O(n^4)$
- $O(n^3)$
- $O(n^2)$
- $O(n)$

14.8.2 Compute the Running Time of a Method

1. What is the running time of the *foo4* method (assume that the parameter *arr* is a two-dimensional array of *n* rows and *n* columns)?

```
public static void foo4( int [ ][ ] arr )
{
  for ( int i = 0; i < arr.length; i++ )
  {
   for ( int j = arr[i].length - 1; j >= 0; j-- )
     System.out.println( "Hello world" );
  }
}
```

   Description

2. What is the running time of the *foo5* method (assume that the parameter *arr* is a three-dimensional array where each dimension has exactly *n* elements)?

```
public static void foo5( int [ ][ ][ ] arr )
{
  for ( int i = 0; i < arr.length; i++ )
  {
   for ( int j = 0; j < arr[i].length; j++ )
   {
    for ( int k = 0; k < arr[i][j].length; k++ )
     System.out.println( "Hello world" );
   }
  }
}
```

   Description

3. What is the running time of the *foo6* method?

```
public static void foo6( int n )
{
  if ( n <= 0 )
     System.out.println( "Hello world" );
  else
     foo6( n - 1 );
}
```

Description

4. What is the running time of the *foo7* method?

```
public static int foo7( int n )
{
  // n is guaranteed to be >= 0
  if ( n == 0 )
      return 0;
  else
      return ( n + foo7( n - 1 ) );
}
```

Description

5. What is the running time of the *foo8* method?

```
public static int foo8( int n )
{
  // n is guaranteed to be >= 1
  if ( n == 1 || n == 2 )
     return 1;
  else
     return ( foo8( n - 1 ) + foo8( n - 2 ) );
}
```

Description

Hint: Note that $T(n - 2) <= T(n - 1)$.

6. What is the running time of the *foo9* method?

```
public static void foo9( int n )
{
  // n is guaranteed to be >= 0
  if ( n == 0 )
     System.out.println( "done" );
  else
     foo9( n / 2 );
}
```

Description

7. What is the running time of the *foo10* method as a function of *n* and *p*?

```
public static void foo10( int n, int p )
{
  // n and p are guaranteed to be >= 1
  if ( p >= n )
    System.out.println( "done" );
  else
    foo10( n, 2 * p );
}
```

Description

8. What is the running time of the *foo11* method?

```
public static void foo11( int n )
{
  // n is guaranteed to be >= 0
  if ( n == 0 )
      return 0;
   else
      return ( 5 + 2 * foo11( n - 1 ) );
 }
```

Description

14.8.3 Programming Projects

1. Write a program that includes a method taking a single-dimensional array of *ints* as its only parameter, and returning the average of all the elements of the array. Add the necessary code to count how many statements are executed in the innermost loop. Run several simulations depending on the number of elements in the parameter integer array. What is the running time of that method as a function of the number of elements of the parameter array?
2. Write a program that includes a method converting a two-dimensional array of *ints* to a two-dimensional array of *boolean* values. If the integer value is greater than or equal to 0, then the corresponding *boolean* value is *true;* otherwise it is *false*. Add the necessary code to count how many statements are executed in the innermost loop. Run several simulations depending on the number of rows and columns in the argument integer array. What is the running time of that method as a function of the number of rows and columns of the parameter array? (You should assume that each row has the same number of columns.)
3. Write a program that includes a method computing the largest element of a given column (represented by a parameter of the method) of a two-dimensional array of *ints*. Add the necessary code to count how many statements are executed in the innermost loop. Run several simulations depending on the number of rows and columns in the parameter integer array, as well as the index of the column for which the method calculates the largest element. Does the running time of the method depend on the column index? the number of rows? the number of columns? What is the running time of that method as a function of the number of rows and columns of the parameter array and the column index? (You should

assume that each row has the same number of columns.)

4. Write a program that includes a method taking a two-dimensional array of *ints* as its only parameter, and returning a single dimensional array of *ints* such that each element of the returned array is the sum of the corresponding row in the parameter array. Add the necessary code to count how many statements are executed in the innermost loop. Run several simulations depending on the number of rows and columns in the parameter integer array. What is the running time of that method as a function of the number of rows and columns of the parameter array? (You should assume that each row has the same number of columns.)
5. Write a program that implements a recursive Binary Search, and add the necessary code to count how many times *binarySearchRecursive* is being called. Run several simulations on arrays of 32, 64, and 128 elements. How many times is the method called in the best-case scenario and worst-case scenario? Does that match our analysis in the chapter?
6. Write a program that implements a recursive method to compute the factorial of a number and add the necessary code to count how many times the method is being called. Run several simulations depending on the value of *n*. How many times is the method called? What is the running time of this method?
7. Write a program that includes a method converting a *String* of 0s and 1s to its equivalent decimal number and add the necessary code to count how many times the method is being called. Run several simulations depending on the length of the input *String*. How many times is the method

called? What is the running time of that method?

8. Write a program that implements Counting Sort to sort an array of integers. Research how Counting Sort works before you implement it. What is the main difference between Counting Sort and other sorting algorithms such as Selection Sort, Bubble Sort, and Insertion Sort? In what situations would Counting Sort be efficient or inefficient to sort an array of integers?

14.8.4 Technical Writing

1. Explain why it is important to consider running time when coding algorithms. Use an example to illustrate your point. Your example, web-based or not, should deal with a lot of data.

14.8.5 Group Project (for a group of 1, 2, or 3 students)

1. The Fibonacci sequence is defined as follows:

$$Fib(1) = Fib(2) = 1$$

$$Fib(n) = Fib(n-1) + Fib(n-2)$$

Description

Write a class with two *static* methods and a *main* method. The two *static* methods compute and return the *n*th Fibonacci number two different ways:

- Recursively using the general case formula and base cases above
- Using a loop (based on the formulas above)

Analyze mathematically the running time of each method as a function of *n*, the only parameter.

Inside the *main* method, call both methods with an argument such that they run for approximately 1 second. What value of *n* did you come up with for each method? What conclusion can you draw from this simulation? Does it validate your mathematical analysis?

© Lois GoBe/Shutterstock.

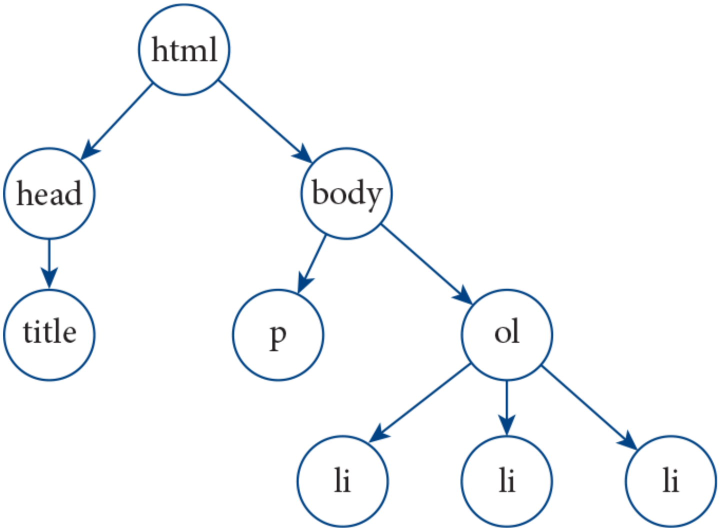
CHAPTER 15

General Trees, Binary Trees, and Binary Search Trees

CHAPTER CONTENTS

Introduction
15.1 Trees
15.2 Binary Trees
15.3 Binary Tree Implementation
15.4 **Programming Activity 1: Working with Binary Trees**
15.5 Binary Search Trees
    15.5.1 Searching in a Binary Search Tree
    15.5.2 Inserting in a Binary Search Tree
    15.5.3 Deleting from a Binary Search Tree
15.6 Binary Search Tree Implementation
15.7 Rebalancing a Binary Search Tree
15.8 **Programming Activity 2: Working with Binary Search Trees**
15.9 Chapter Summary
15.10 Exercises, Problems, and Projects

15.10.1 Multiple Choice Exercises
15.10.2 Reading and Understanding Code
15.10.3 Fill In the Code
15.10.4 Identifying Errors in Code
15.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
15.10.6 Write a Short Program
15.10.7 Programming Projects
15.10.8 Technical Writing
15.10.9 Group Project



Introduction

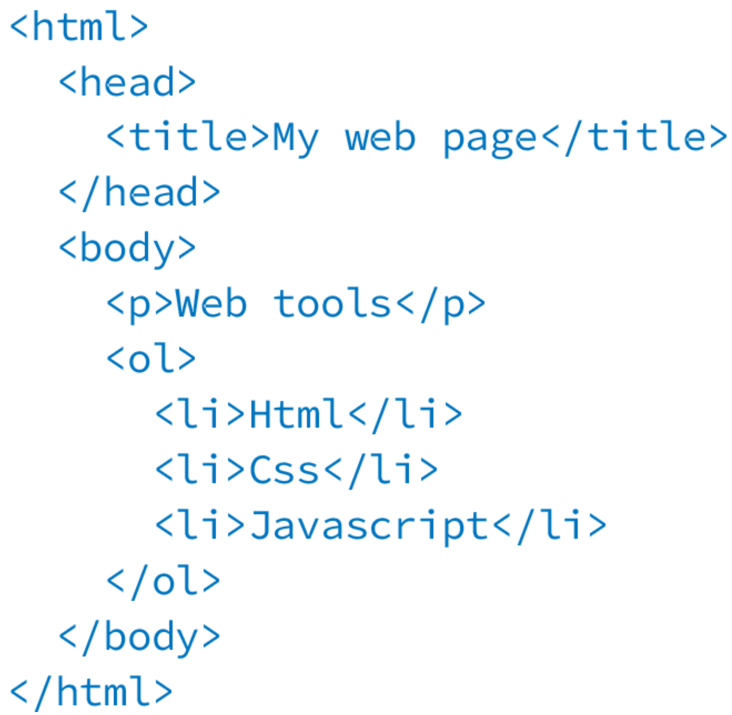
A **tree** is a data structure simulating the hierarchical nature of a tree, including a root and its descendants. As in linked lists, data is stored in nodes; nodes are connected via a parent-child relationship. Trees are used in many software applications, like database management software, and client-side web development.

Ideally, the objective is to provide an abstract data structure that is compact and efficient to search, insert into, modify, and delete from.

Relational database tables can be visualized as two-dimensional tables, but are stored in memory in B-trees, which guarantee efficient inserting, deleting and searching at all times.

In web programming, the **document object model (DOM)** specifies a tree structure that an HTML document is converted to when loaded in memory. Any change to the DOM tree is reflected on the web page by the browser. For example, the following HTML code will display a simple web page with a paragraph and an ordered list of three items (**Example 15.1**):

Example 15.1 HTML Code for a Simple Web Page



```
<html>
  <head>
    <title>My web page</title>
  </head>
  <body>
    <p>Web tools</p>
    <ol>
      <li>Html</li>
      <li>Css</li>
      <li>Javascript</li>
    </ol>
  </body>
</html>
```

Description

When a web browser receives that HTML page, it converts it to the following tree (**Figure 15.1**):

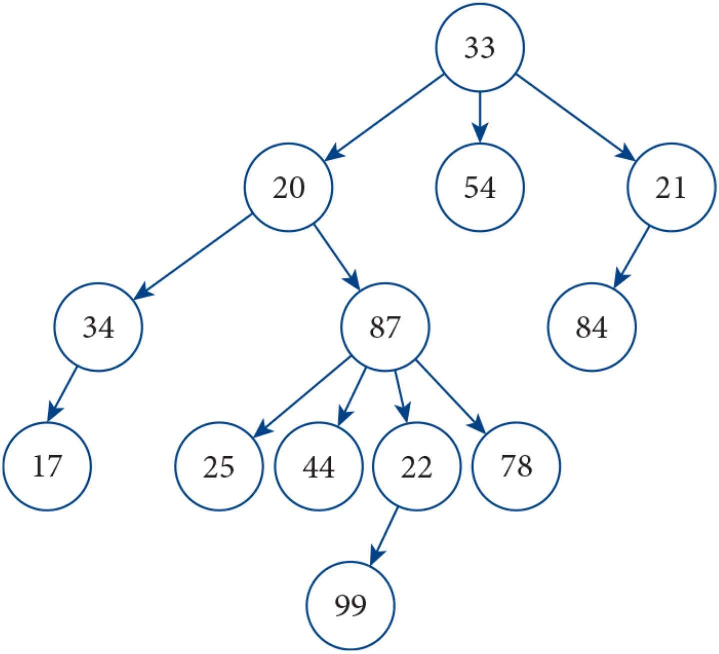
Figure 15.1 The Tree for the HTML Code of Example 15.1

Description

By navigating the tree, we can retrieve elements of the web page—for example, the paragraph or any of the list elements above—modify them, style them, delete them, or insert new elements. Any change to the DOM tree will trigger an update of the web page displayed by the browser.

REFERENCE POINT

You can get more information about HTML at https://www.w3.org/TR/html/. To get more information about the Document Object Model (DOM), visit https://www.w3.org/DOM/DOMTR



15.1 Trees

Figure 15.2 shows a tree storing integers. The top node in the tree is called the **root**. The root has children nodes, its children nodes have children nodes themselves, ...

Figure 15.2 A Tree Storing Integers

Description

Nodes are connected with **directed edges**. A directed edge defines a **parent-child** relationship between two nodes. A node can have 0 or more children but can only have one parent maximum. Every node in the tree except the root has exactly one parent. The node containing 87 has four children. The parent of the node containing 87 is the node containing 20. Nodes that have the same parent are called **siblings;** thus, the nodes containing 25, 44, 22, and 78 are siblings.

A node with no child is called a **leaf;** a node that is not a leaf is called an **interior** node. The node containing 33 is the root.

The other interior nodes are the nodes containing 20, 34, 87, 22, and 21. The nodes containing 17, 25, 44, 99, 78, 54, and 84 are leaves.

The **degree** of a node is its number of children. The degree of the node containing 20 is 2; the degree of the node containing 87 is 4.

The **depth,** or **level,** of a node is the number of links, or edges, from the root to that node. The nodes containing 20, 54, and 21 have a depth of 1; they are at level 1. The nodes containing 34, 87, and 84 have a depth of 2; they are at level 2.

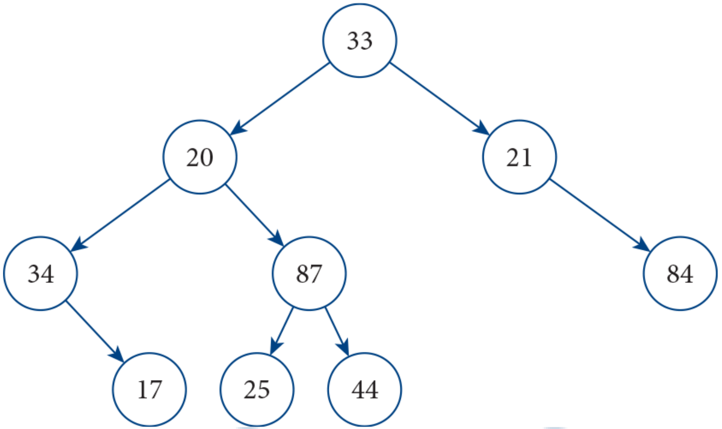
The **height** of a node is the largest number of edges from that node to a leaf. The node containing 87 has height 2. The node containing 21 has height 1; the node containing 33 has height 4. The height of the tree is the height of its root, 4 in this example.

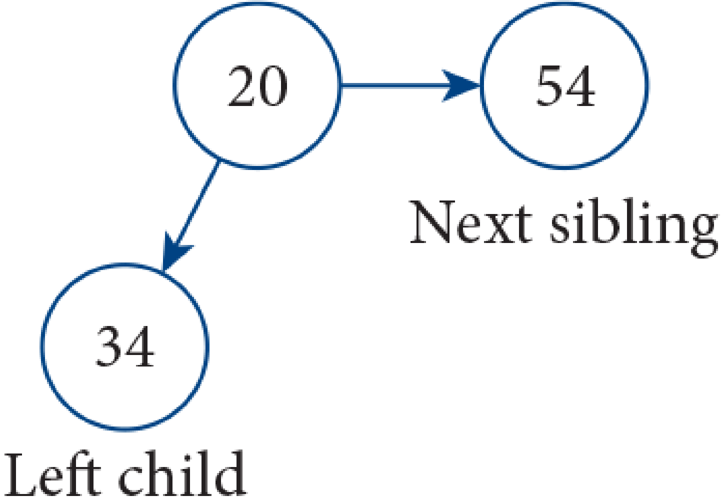
Table 15.1 summarizes all these definitions.

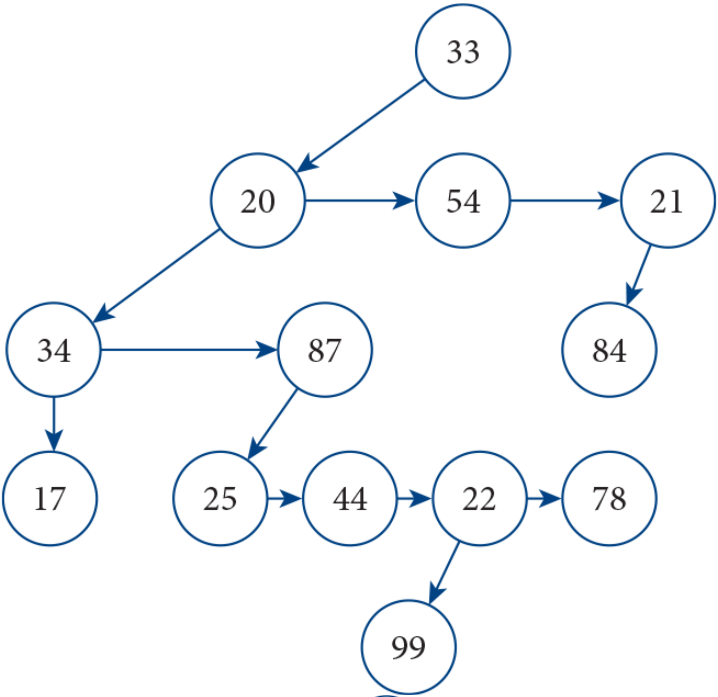
Tree Terminology

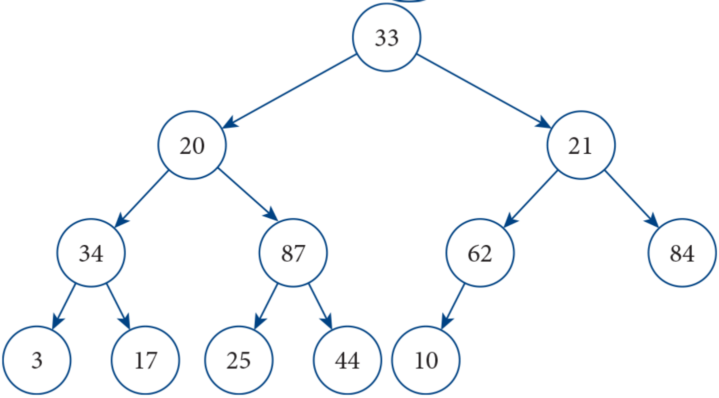
Term	Description
Leaf	A node with no child.
Interior node	A node with at least one child.
Siblings	Two (or more) nodes with the same parent.
Degree of a node	The number of children of a node.
Depth of a node	The number of edges on the path from the root to the node.
Level of a node	The number of edges on the path from the root to the node.
Height of a node	The largest number of edges on a path from a node to a leaf.
Height of the tree	The height of the root.

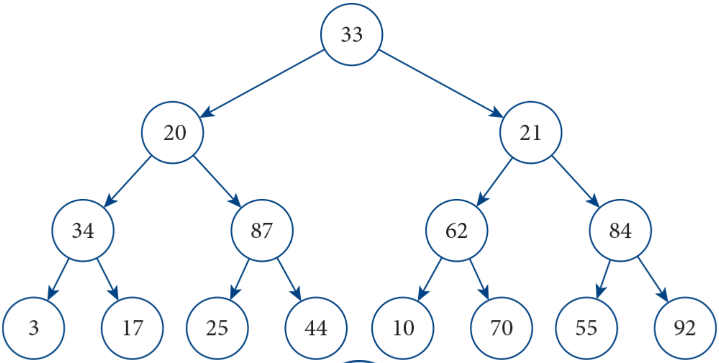
Description

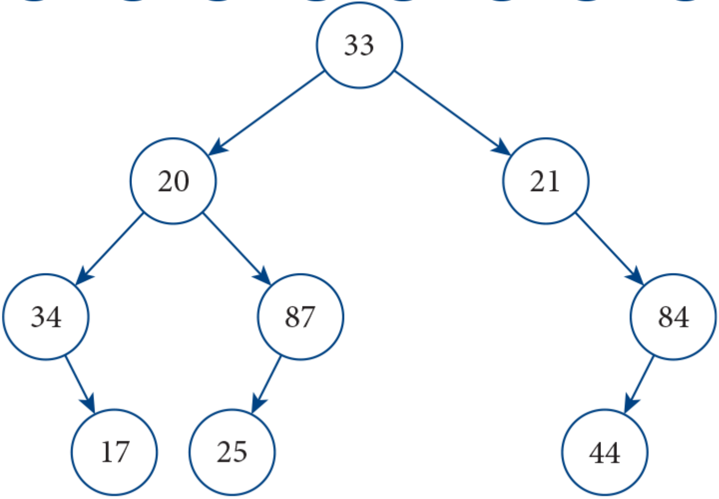



Next sibling
Left child






33
20
21
34
87
62
84
3
17
25
44
10
70
55
92


33
20
21
34
87
84
17
25
44

15.2 Binary Trees

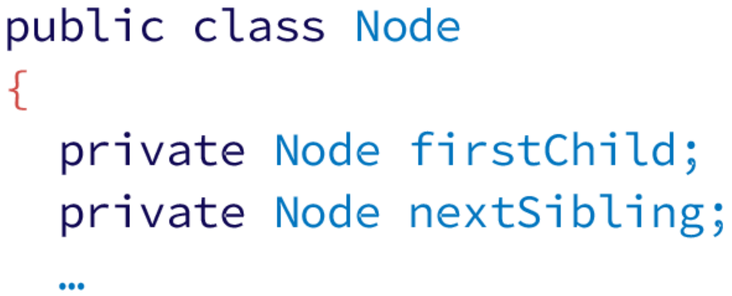
A *binary tree* is a tree whose nodes have degree 0, 1, or 2; i.e., they have either 0, 1, or 2 children. **Figure 15.3** shows a binary tree storing integers. All its nodes have degree 0, 1, or 2. The nodes containing 17, 25, 44, and 84 have degree 0; they are leaves. The nodes containing 34 and 21 have degree 1. The nodes containing 33, 20, and 87 have degree 2.

Figure 15.3 A Binary Tree Storing Integers

Description

Every general tree can be converted to a binary tree. In order to do that, we can define a *Node* as having two *Node* children (they may or may not be *null):* a first child and a next sibling. **Example 15.2** shows how such a *Node* class can be coded. **Figure 15.4** shows an example of such a node corresponding to the node containing 20 in Figure 15.2.

Example 15.2 A *Node* Class That Can Be Used to Convert a Regular Tree to a Binary Tree



```
public class Node
{
  private Node firstChild;
  private Node nextSibling;
  ...
```

Figure 15.4 An Example of a First Child – Next Sibling Node

Description

Using that type of node, any general tree can be represented and stored as a binary tree. **Figure 15.5** shows the general tree of Figure 15.2 represented and stored as a binary tree.

Figure 15.5 The Regular Tree from Figure 15.2 Represented as a Binary Tree Using First Child – Next Sibling Nodes

Description

Generally, it is desirable for a binary tree to have minimal height, so that it is as compact as possible. Later in the chapter, we explain why in great detail.

A tree that has all its levels completely full and its last level filled from left to right is called a **complete binary tree.** An example is shown in **Figure 15.6**. A tree that has all its levels completely full is called a **full binary tree.** An example is shown in **Figure 15.7**. A full binary tree is a special case of a complete binary tree; a complete binary tree is generally not a full binary tree.

Figure 15.6 A Complete Binary Tree

Description

Figure 15.7 A Full Binary Tree

Description

Binary trees that are either full, complete, or close to complete can be very compact data structures. A lot of nodes can be stored in a tree that has very low height. **Table 15.2** shows the number of nodes per level in a full binary tree (remember that in a full binary tree, every level is full). As we go down one level at a time in the binary tree, every level contains two times the number of nodes in its previous level. Thus, at level *L*, we have 2^L nodes.

Number of Nodes per Level in a Full Binary Tree

Level	Number of Nodes
0	$1 = 2^0$
1	$2 = 2^1$
2	$4 = 2^2$
3	$8 = 2^3$
4	$16 = 2^4$
5	$32 = 2^5$
6	$64 = 2^6$
L	2^L

Description

If we have *n* nodes in a binary tree whose deepest level is *L*, what is the value of *L* as a function of *n*?

$$n = \text{total \# of nodes} = \text{sum of nodes for all levels} = 1 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + \ldots + 2^L$$

Description

This is a geometric series of ratio 2 and is equal to $(2^{L+1} - 1)/(2 - 1) = 2^{L+1} - 1$. This formula can easily be proven by induction. The following is another simple proof:

$$n = 1 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + \ldots + 2^L$$

Description

Multiplying each side by 2, we get:

$$2n = 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + \ldots + 2^L + 2^{L+1}$$

Description

Thus, subtracting *n* from *2n,* most terms cancel out and we have:

$$2n - n = 2^{L+1} - 1; \text{thus, } n = 2^{L+1} - 1, \text{or } n + 1 = 2^{L+1}$$

Description

Taking the log of both sides:

$$\log(n+1) = L + 1, \text{or } L = \log(n+1) - 1$$

Description

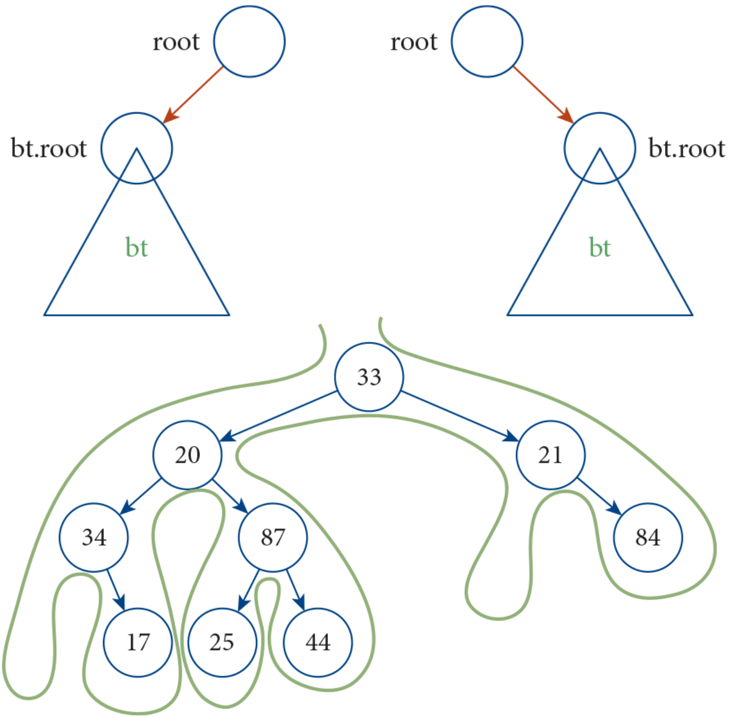
Thus, *L* is very small in comparison to *n.* If we store the emails of all the Facebook users, i.e., over 1 billion emails, in a full (or complete) binary tree, we only need 30 levels in the binary tree (1 billion is roughly equal to 2^{30}, and therefore the log of 1 billion is roughly equal to 30).

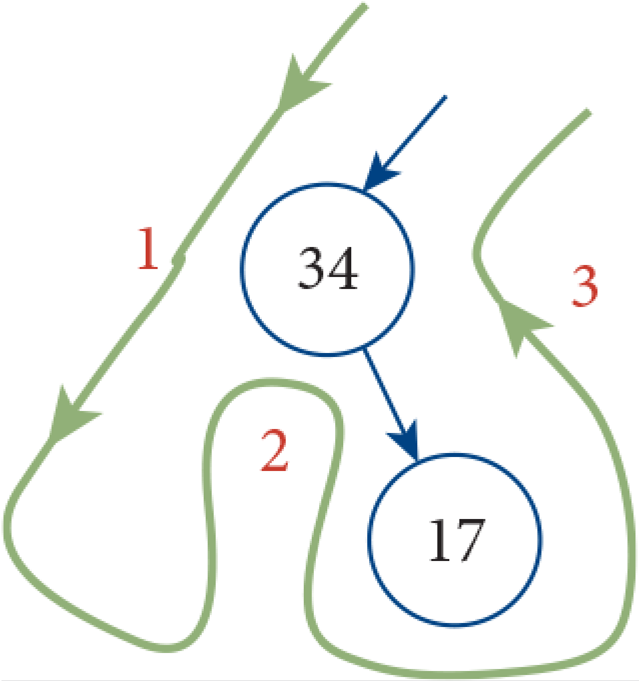
Any node in a tree can be considered the root of a tree, the tree comprised of itself and its descendants. The tree rooted at the left child of a node is considered its **left subtree.** The tree rooted at the right child of a node is considered its **right subtree.** For the tree in Figure 15.3, the “root” of the left subtree is the node containing 20; the “root” of the right subtree is the node containing 21.

For many types of binary trees, it is very desirable that the height of a tree is minimal or close to it. A node is **height-balanced** if the difference in height between its left and right subtrees (or vice-versa) is less than or equal to 1. A binary tree is height-balanced if all its nodes are height-balanced; an empty tree is height-balanced. The binary trees in Figure 15.3, 15.6, and 15.7 are all height-balanced. The binary tree in **Figure 15.8** is not height-balanced because the node containing 21 is not height-balanced.

Figure 15.8 A Binary Tree That Is Not Height-Balanced

Description


root
bt.root
bt
root
bt.root
bt



```
Pre order: 33 20 34 17 87 25 44 21 84
In order: 34 17 20 25 87 44 33 21 84
Post order: 17 34 25 44 87 20 84 21 33
```

15.3 Binary Tree Implementation

As with linked lists, we use a *Node* class as a tool when implementing a binary tree. A *Node* stores data and links or references to its two children. The data contained in a *Node* can be of a basic data type or a user-defined data type; we can also use generics for the most flexible implementation. To keep things simple and focus on the functionality of binary trees, we store integers in *IntegerNode* objects. In addition to the data, an *IntegerNode* has a left child, an *IntegerNode,* and a right child, another *IntegerNode.* **Example 15.3** shows the *IntegerNode* class, a class encapsulating a node storing an *int.* In addition to the instance variables, it includes a constructor, accessors, and mutators. The constructor (lines 11–18) constructs an *IntegerNode* storing a given value, its parameter; its left and right children are *null.* The *IntegerNode* class is simply a utility class to store an *int* and links to other nodes in the binary tree.

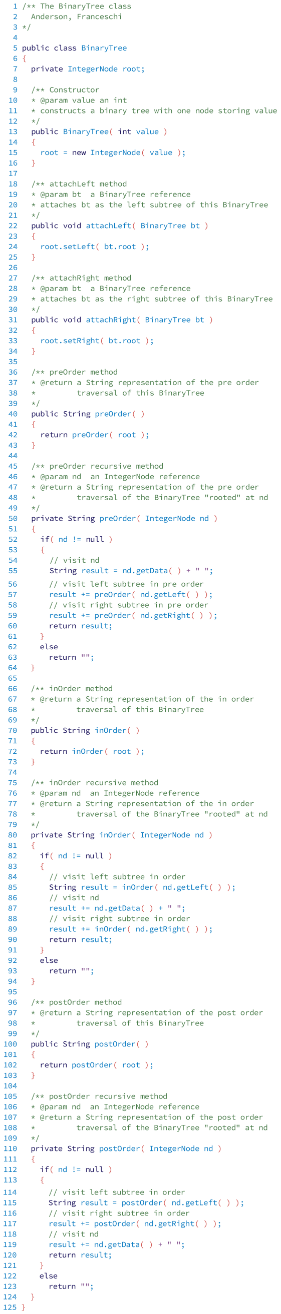
Example 15.3 The *IntegerNode* Class

```java
/** The IntegerNode class
  Anderson, Franceschi
*/

public class IntegerNode
{
  private int data;
  private IntegerNode left;
  private IntegerNode right;

  /** Constructor
  * @param data
  * initializes data instance variable with data
  */
  public IntegerNode( int data )
  {
     setData( data );
  }

  /** Accessor for data
  * @return data
  */
  public int getData( )
  {
     return data;
  }

  /** Accessor for left
  * @return left
  */
  public IntegerNode getLeft( )
  {
     return left;
  }

  /** Accessor for right
  * @return right
  */
  public IntegerNode getRight( )
  {
     return right;
  }

  /** Mutator for data
  * @param data    an int, the new value for data
  * @return        a reference to this object
  */
  public IntegerNode setData(int data )
  {
     this.data = data;
     return this;
  }

  /** Mutator for left
  * @param left a Node, the new Node reference for left
  * @return     a reference to this object
  */
  public IntegerNode setLeft(IntegerNode left )
  {
     this.left = left;
     return this;
  }

  /** Mutator for right
  * @param right a Node, the new Node reference for right
  * @return      a reference to this object
  */
  public IntegerNode setRight( IntegerNode right )
  {
     this.right = right;
     return this;
  }
}
```

Description

Example 15.4 shows the *BinaryTree* class. We only need one instance variable (line 7), *root,* an *IntegerNode,* the root of the binary tree. From *root,* we can access its left and right children, their left and right children, ...

Example 15.4 The *BinaryTree* Class



```java
/** The BinaryTree class
  Anderson, Franceschi
*/

public class BinaryTree
{
  private IntegerNode root;

  /** Constructor
  * @param value an int
  * constructs a binary tree with one node storing value
  */
  public BinaryTree( int value )
  {
    root = new IntegerNode( value );
  }

  /** attachLeft method
  * @param bt  a BinaryTree reference
  * attaches bt as the left subtree of this BinaryTree
  */
  public void attachLeft( BinaryTree bt )
  {
    root.setLeft( bt.root );
  }

  /** attachRight method
  * @param bt  a BinaryTree reference
  * attaches bt as the right subtree of this BinaryTree
  */
  public void attachRight( BinaryTree bt )
  {
    root.setRight( bt.root );
  }

  /** preOrder method
  * @return a String representation of the pre order
  *         traversal of this BinaryTree
  */
  public String preOrder( )
  {
    return preOrder( root );
  }

  /** preOrder recursive method
  * @param nd  an IntegerNode reference
  * @return a String representation of the pre order
  *         traversal of the BinaryTree "rooted" at nd
  */
  private String preOrder( IntegerNode nd )
  {
    if( nd != null )
    {
      // visit nd
      String result = nd.getData( ) + " ";
      // visit left subtree in pre order
      result += preOrder( nd.getLeft( ) );
      // visit right subtree in pre order
      result += preOrder( nd.getRight( ) );
      return result;
    }
    else
      return "";
  }

  /** inOrder method
  * @return a String representation of the in order
  *         traversal of this BinaryTree
  */
  public String inOrder( )
  {
    return inOrder( root );
  }

  /** inOrder recursive method
  * @param nd  an IntegerNode reference
  * @return a String representation of the in order
  *         traversal of the BinaryTree "rooted" at nd
  */
  private String inOrder( IntegerNode nd )
  {
    if( nd != null )
    {
      // visit left subtree in order
      String result = inOrder( nd.getLeft( ) );
      // visit nd
      result += nd.getData( ) + " ";
      // visit right subtree in order
      result += inOrder( nd.getRight( ) );
      return result;
    }
    else
      return "";
  }

  /** postOrder method
  * @return a String representation of the post order
  *         traversal of this BinaryTree
  */
  public String postOrder( )
  {
    return postOrder( root );
  }

  /** postOrder recursive method
  * @param nd  an IntegerNode reference
  * @return a String representation of the post order
  *         traversal of the BinaryTree "rooted" at nd
  */
  private String postOrder( IntegerNode nd )
  {
    if( nd != null )
    {
      // visit left subtree in order
      String result = postOrder( nd.getLeft( ) );
      // visit right subtree in order
      result += postOrder( nd.getRight( ) );
      // visit nd
      result += nd.getData( ) + " ";
      return result;
    }
    else
      return "";
  }
}
```

Description

The constructor (lines 9–16) constructs a binary tree storing one element, *value,* its parameter. Its root node contains *value* (line 15).

At this point, our binary tree has no particular ordering. Thus, in order to add elements to our binary tree, we provide methods that attach a binary tree to this binary tree (lines 18–25 and 27–34).

The *attachLeft* method (lines 18–25) takes a *BinaryTree* parameter (that contains 0 or more elements) and attaches its *root* as the left child of *root* (line 24). The *attachRight* method (lines 27–34) takes a *BinaryTree* parameter (that contains 0 or more elements) and attaches its *root* as the right child of *root* (line 33). **Figure 15.9** shows what the *attachLeft* and *attachRight* methods do.

Figure 15.9 The *attachLeft* and *attachRight* Methods

Description

In addition to the methods above, we provide methods that return a *String* listing all the elements stored in the binary tree. Going through all the elements in the binary tree is called **traversing** the tree. That gives us a human-readable representation of the tree, and that is what the *toString* method usually does. However, we can traverse the binary tree several ways, so we do not have one *toString* method, but several traversal methods.

We typically can traverse the binary tree in level order, pre order, in order, or post order.

Traversing the tree in level order, we start at the root (level 0), and visit all the nodes one level at a time from top to bottom, going left to right at each level. Thus, we first visit the root (level 0), then visit its left child and right child (level 1), then the nodes at level 2 from left to right, then the nodes at level 3 from left to right, and so on.

Traversing in pre order, in order, and post order involve walking around the tree counterclockwise, as shown in **Figure 15.10**, visiting one node at a time. **Figure 15.11** shows how we visit the node containing 34. Even if the left child of a node is *null,* it is considered a visit. We will “see” 34 a total of three times: two times before seeing 17 and one time after it. Traversing in pre order, we list the element the first time we see it. Traversing in order, we list the element the second time we see it. Traversing in post order, we list the element the last (third) time we see it.

Figure 15.10 Pre Order, In Order, and Post Order Traversals Are Performed Walking Counterclockwise Around the Tree

Description

Figure 15.11 We Can Visit a Node on Three Occasions: 1 → Pre Order, 2 → In Order, 3 → Post Order

Description

Thus, when traversing the tree either in pre order or in order, we will list 34 before we list 17. When traversing the tree in post order, we will list 17 before we list 34.

Table 15.3 lists how each traversal works and what the results are for the tree in Figure 15.10.

Tree Traversal Strategies and Results for the Binary Tree in Figure 15.10

Traversal	Strategy	Result
Level Order	Visit one level at a time, left to right	33 20 21 34 87 84 17 25 44
Pre Order	Visit node the first time we see it	33 20 34 17 87 25 44 21 84
In Order	Visit node the second time we see it	34 17 20 25 87 44 33 21 84
Post Order	Visit node the third/last time we see it	17 34 25 44 87 20 84 21 33

Description

COMMON ERROR TRAP

When traversing a binary tree in order, if a node has no left child, we will see that node two times before seeing its right child. Thus, its data will be listed before the data of its right child or its right child’s descendants when traversing the binary tree in order.

The level order traversal is more complicated to code than the three other traversals and one of the topics of Programming Activity 1. The pre order, in order, and post order traversals are well suited for a recursive implementation.

The recursive algorithm for the pre order traversal is as follows:

- List the content of the current node.
- Traverse in pre order the left subtree, i.e., the binary tree rooted at the left child of the current node.
- Traverse in pre order the right subtree, i.e., the binary tree rooted at the right child of the current node.

As described above, we make two recursive calls when traversing a binary tree, one on the left subtree and one on the right subtree. Thus, we need a way to programmatically define a subtree. Like a binary tree is fully defined by its root, a subtree is fully defined by one node, its “root.” For example, the root of the left subtree is the left child of *root.* Thus, our recursive pre order traversal method has this method header (line 50):

```
private String preOrder( IntegerNode nd )
```

Description

It traverses in pre order the subtree rooted at *nd* and returns a *String* representation of that traversal.

To traverse in pre order the whole binary tree, i.e., the “subtree” rooted at *root,* we simply call the method above and pass *root* as its argument. However, from a client class, we do not have access to the *root* instance variable of a binary tree and therefore cannot call that method. We define this method as *private* and provide a *public* “wrapper” method (lines 36–43) that takes no parameter and calls the *private* recursive method (line 42) in order to traverse the whole binary tree.

The base case of the recursive method happens when the subtree that we traverse is empty, i.e., *nd* is *null* (line 62); in this case, it returns an empty *String* (line 63). If the subtree is not empty (line 52), we place the data in *nd* in the *String result* (line 55), then concatenate to *result* the *String* returned by the pre order traversal of the left subtree (lines 56–57), and finally concatenate to *result* the *String* returned by the pre order traversal of the right subtree (lines 58–59). At line 60, we return *result.*

The in order and post order traversal methods (lines 66–94 and 96–124) are coded using the same pattern as the *preOrder* method.

The recursive algorithm for the in order traversal is as follows:

- Traverse in order the left subtree, i.e., the binary tree rooted at the left child of the current node.
- List the content of the current node.
- Traverse in order the right subtree, i.e., the binary tree rooted at the right child of the current node.

  The recursive algorithm for the post order traversal is as follows:
- Traverse in post order the left subtree, i.e., the binary tree rooted at the left child of the current node.
- Traverse in post order the right subtree, i.e., the binary tree rooted at the right child of the current node.
- List the content of the current node.

Example 15.5 shows how we can use the *attachLeft* and *attachRight* methods to build the binary tree in Figure 15.10. We traverse the binary tree at lines 29–31. The output, shown in **Figure 15.12**, matches the traversals listed in Table 15.3.

Example 15.5 A Client of the *BinaryTree* Class

```java
/** The BinaryTreeClient class
   Anderson, Franceschi
 */
 public class BinaryTreeClient
 {
   public static void main( String [] args )
   {
     BinaryTree bt33 = new BinaryTree( 33 );
     BinaryTree bt20 = new BinaryTree( 20 );
     BinaryTree bt21 = new BinaryTree( 21 );
     BinaryTree bt34 = new BinaryTree( 34 );
     BinaryTree bt87 = new BinaryTree( 87 );
     BinaryTree bt84 = new BinaryTree( 84 );
     BinaryTree bt17 = new BinaryTree( 17 );
     BinaryTree bt25 = new BinaryTree( 25 );
     BinaryTree bt44 = new BinaryTree( 44 );

     bt33.attachLeft( bt20 );
     bt33.attachRight( bt21 );

     bt20.attachLeft( bt34 );
     bt20.attachRight( bt87 );
     bt21.attachRight( bt84 );

     bt34.attachRight( bt17 );
     bt87.attachLeft( bt25 );
     bt87.attachRight( bt44 );

     System.out.println( "Pre order: " + bt33.preOrder( ) );
     System.out.println( "In order: " + bt33.inOrder( ) );
     System.out.println( "Post order: " + bt33.postOrder( ) );
  }
}
```

Description

Figure 15.12 The Output of Example 15.5

Description

15.4 Programming Activity 1: Working with Binary Trees

In this activity, you will add methods to the *BinaryTree* class to return the leftmost data in the binary tree, count the number of nodes in the tree, count the number of leaves in the tree, calculate the height of the tree, and traverse the tree in level order.

Instructions

Copy the source files in the Programming Activity 1 folder for this chapter to a folder on your computer. Note that all files should be in the same folder.

Inside the *BinaryTree* class, you need to modify and code the following methods (the binary tree stores integer values):

1. A method that returns the leftmost data in the binary tree; this method throws an exception if the binary tree is empty
2. A method that returns the number of nodes in the binary tree
3. A method that returns the number of leaves in the binary tree
4. A method that returns the height of the binary tree
5. A method that returns a *String* listing all the values stored in the binary tree using a level traversal of the binary tree

Note: The data returned by these methods must **not** be kept and maintained using an instance variable (for example, we do not keep track of the number of items in the tree).

Hint: You can use a queue in method # 5. The *ArrayList* class includes methods that you can use to provide the functionality of a queue.

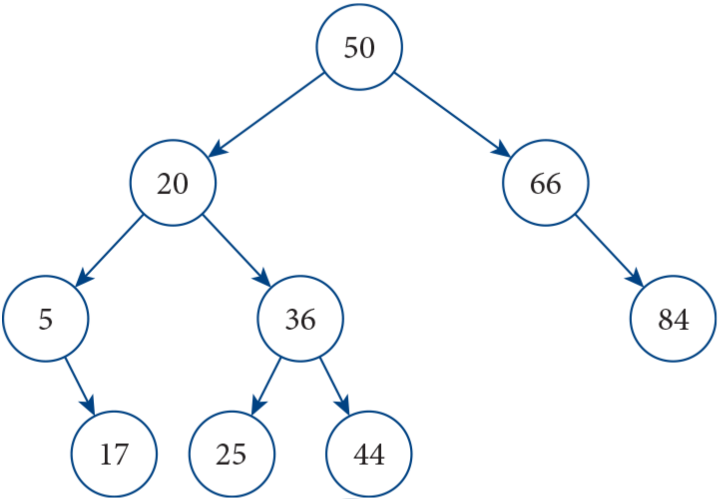
In the client of the class, build a tree storing at least eight values and test all the methods.

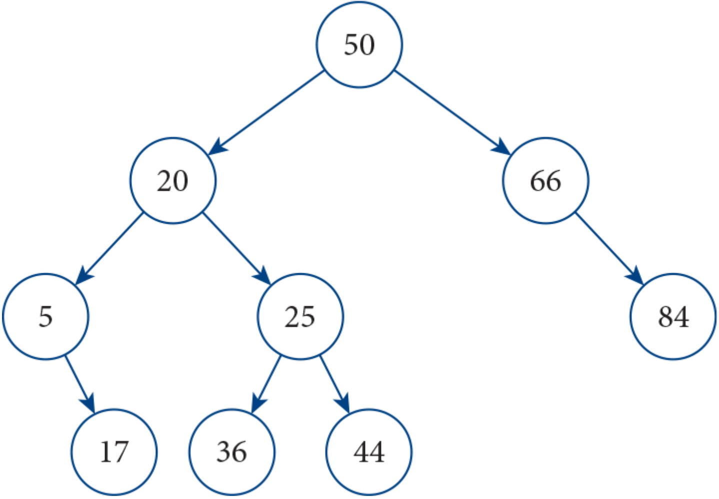
Troubleshooting

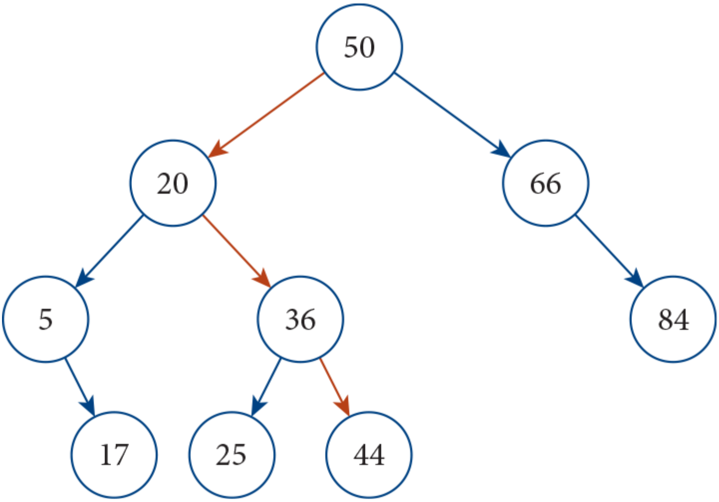
- Code one method at a time and test your results.
- Draw the tree on a piece of paper and check all your results.
- If one of your methods shows a stackoverflow error at run time, check that you have properly coded the base case.

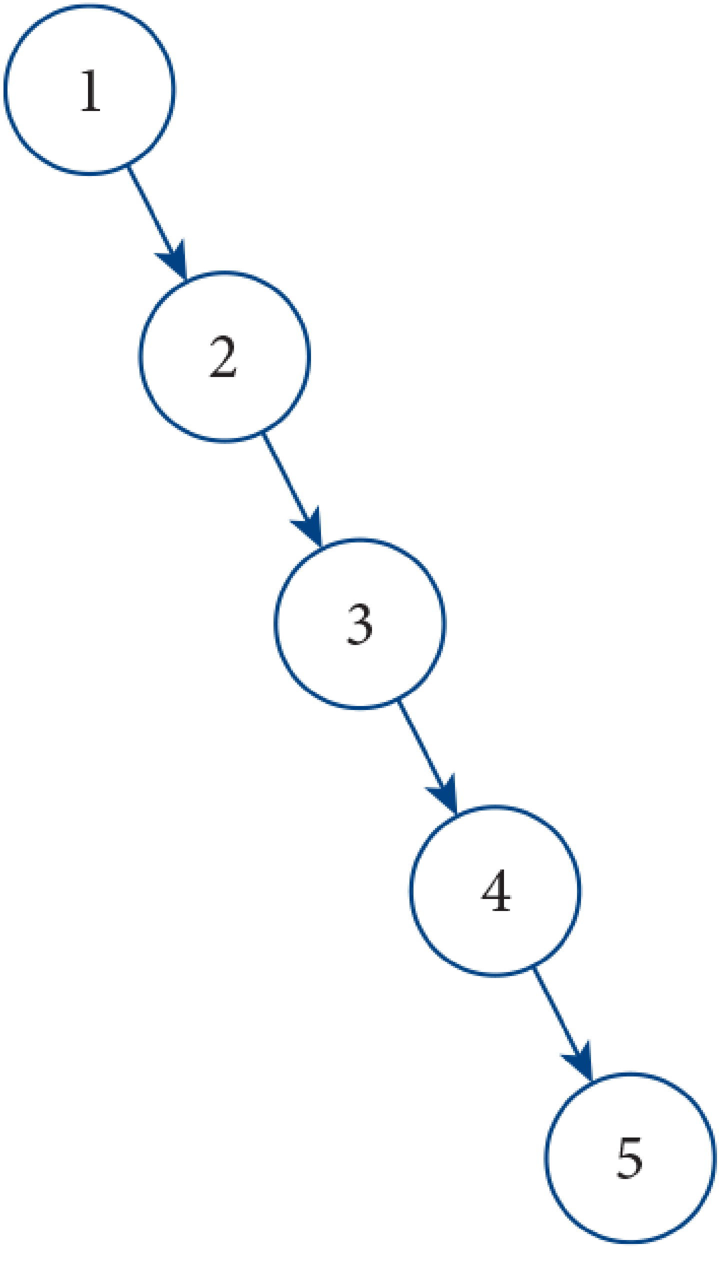
DISCUSSION QUESTIONS

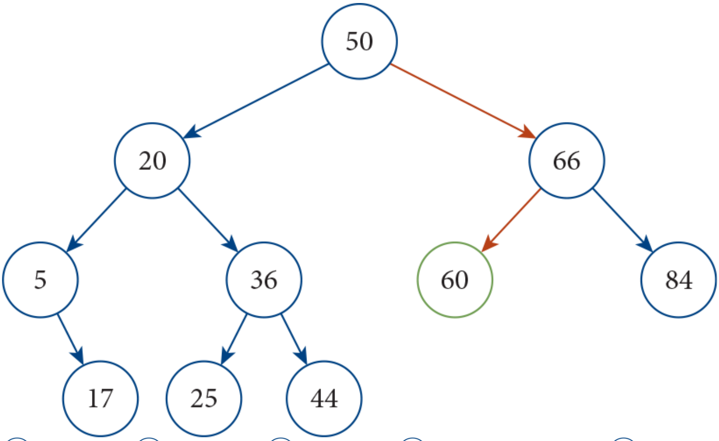
1. **Which methods did you implement recursively? Why did you choose to implement recursively?**
2. **How is the level traversal implementation different from the pre order, in order, and post order implementations?**

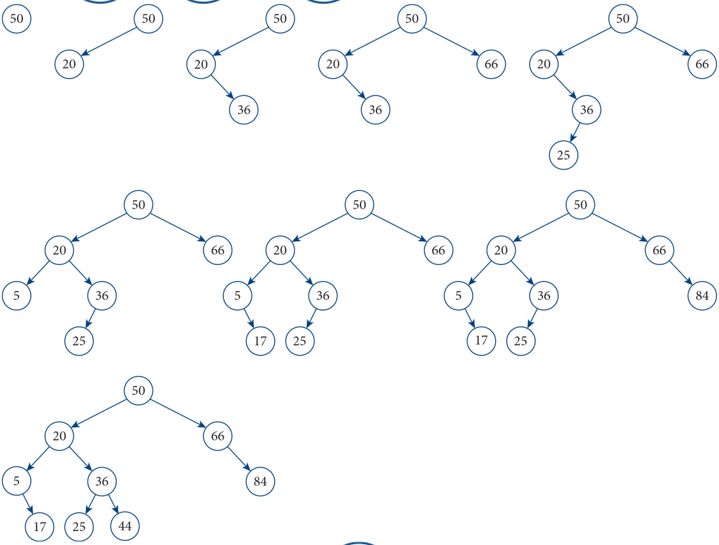


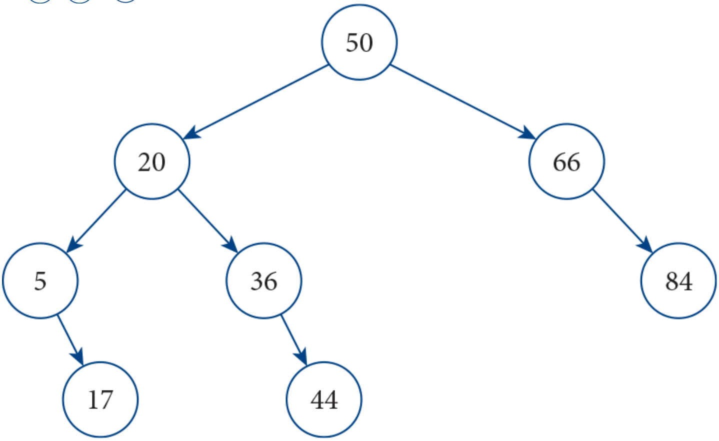


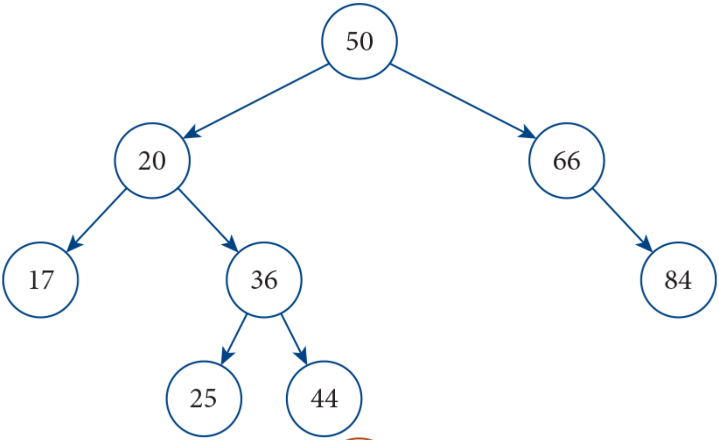


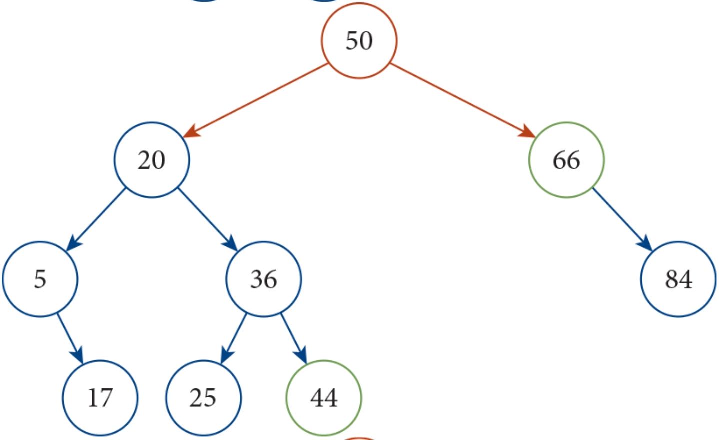


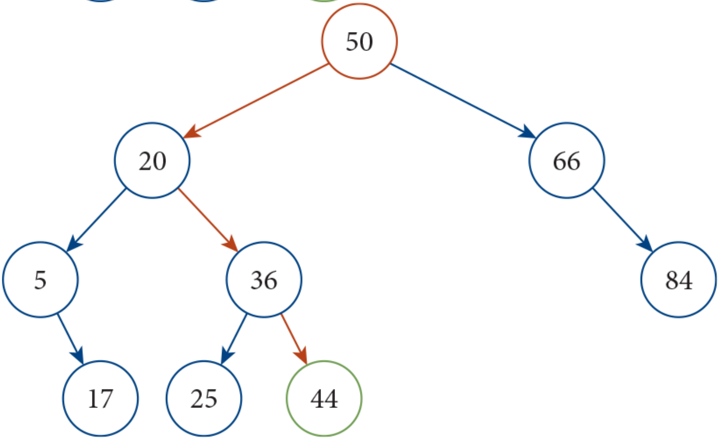


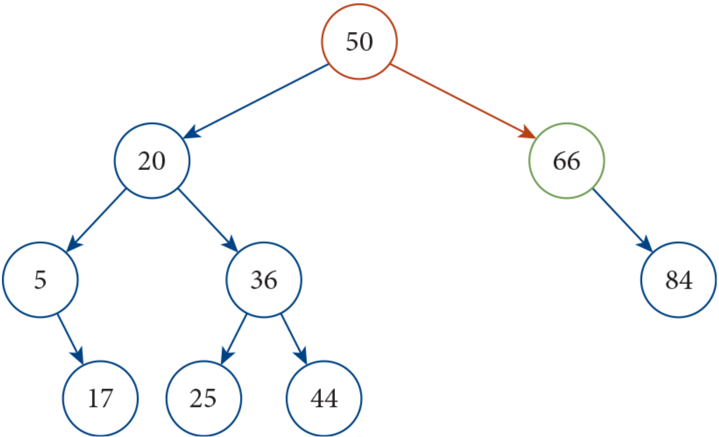


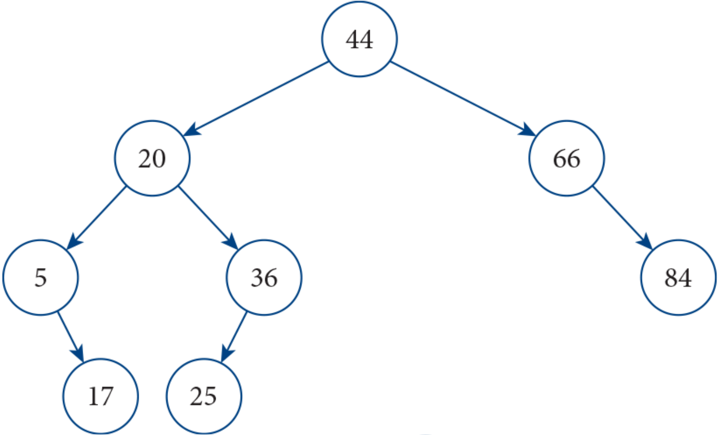


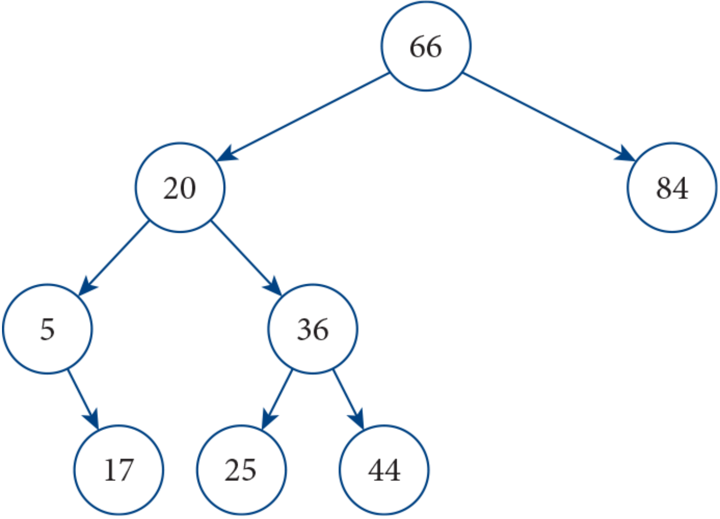












15.5 Binary Search Trees

As we learned earlier, binary trees can be very compact structures. However, without any ordering, binary trees are not suited for searching; we typically need to traverse the whole tree when looking for a particular value. Binary search trees provide a solution to the inefficient searching of a binary tree.

A *binary search tree,* a special type of binary tree, is defined as follows:

- All the elements to the left of the root have a value smaller than the root's value,
- All the elements to the right of the root have a value larger than the root's value,
- The left subtree of a binary search tree is a binary search tree itself, and
- The right subtree of a binary search tree is a binary search tree itself.
- An empty binary tree is a binary search tree.

As we can see, the above definition is recursive (points 1, 2, 3, and 4 define the general case; point 5 defines the base case). **Figure 15.13** shows an example of a binary search tree. Look at the left subtree in which the "root" value is 20; it is a binary search tree itself. The values on its left are all smaller than 20, and the values on its right are all larger than 20. Then, its left and right subtrees are also binary search trees. The left subtree, the "root" value of which is 5, is a binary search tree: its left subtree is empty, 17 is larger than 5, and its right subtree is made up of just one element and is a binary search tree, ...

Figure 15.13 A Binary Search Tree: The values on the left are smaller than 50; the values on the right are larger than 50; both the left and right subtrees are binary search trees themselves.

Description

If we switch 36 and 25 (**Figure 15.14**), we no longer have a binary search tree. Although the values on the left of the root are all smaller than 50, the subtree rooted at the node that contains 25 is not a binary search tree itself, because 36 is on the left of 25 and should be on the right of 25.

Figure 15.14 A Binary Tree That Is NOT a Binary Search Tree: 36 is on the left of 25, a binary search tree property violation.

Description

COMMON ERROR TRAP

If you think you are building a binary search tree and the values do not come out in ascending order when doing an in order traversal of the tree, then the tree is not a binary search tree.

A remarkable property of binary search trees is that values come out in ascending order when doing an in order traversal. For the binary search tree in Figure 15.13, the in order traversal is: 5 17 20 25 36 44 50 66 84.

15.5.1 Searching in a Binary Search Tree

Searching in a binary search tree is similar to doing a binary search. At each comparison, we eliminate one side of the tree. **Figure 15.15** shows how we can search for 44 in our binary search tree. 44 is smaller than 50, so we search left; 44 cannot be found on the right subtree because all the elements on the right of 50 are larger than 50 and therefore larger than 44. Then we look at 20 and since 44 is larger than 20, we search to the right of 20; 44 is larger than 36, so we search right of 36. The next node contains 44. If we search for 43, we follow the same strategy. We first look at 50 and go left; we then look at 20 and go right. We look at 36 and go right, then look at 44 and go left. The node containing 44 does not have a left child, so we conclude that we will not find 43 in the binary search tree of Figure 15.13.

Figure 15.15 Searching for 44 in a Binary Search Tree

Description

Let's make the assumption that a binary search tree is complete. In the worst-case scenario, whether we find the value we are looking for in a leaf or do not find it, we go from the root to a leaf using a path whose length is the height of the tree. Since the height of a complete binary tree is log(*n* +1) − 1 where *n* is the number of nodes in the tree, searching takes O(log *n*). Thus, if we store all the Facebook users and order them using their emails in a complete binary search tree, it will only take us 30 or 31 comparisons in the worst case to find the user we are looking for when we search for an email value.

However, the binary search tree may not be complete; in fact, in the worst-case scenario, it could look like a linked list, as shown in **Figure 15.16**. In this case, searching can take up to *n* steps if we are searching for the largest value in the tree or a value larger than the largest value in the tree. Thus, it is very desirable to keep the binary search tree height-balanced. Later in this chapter, we discuss how we can rebalance a binary search tree so that it is height-balanced.

Figure 15.16 A Binary Search Tree That Looks Like a Linked List

Description

15.5.2 Inserting in a Binary Search Tree

A binary search tree must also support insert and delete operations. After inserting or deleting, the tree must still be a binary search tree afterwards. We also need to keep those operations relatively simple so that they can be easily coded and have an efficient running time.

We typically insert "as a leaf" in a binary search tree. In terms of resetting the links between nodes, it keeps things very simple. There is only one place to insert "as a leaf" in a binary search tree. Furthermore, if we insert in a height-balanced binary search tree, insert takes O(log *n*). Let's insert 60 in the binary search tree of Figure 15.13. In order to find the place to insert, we follow the same strategy as for searching. Since 60 is larger than 50, it will be on the right of 50. Since it is smaller than 66, it will be on the left of 66. The node containing 66 does not have a left child, so we just found the place to insert 60, and that place is unique within the tree (assuming we insert "as a leaf"). **Figure 15.17** visualizes the insertion of 60 in our binary search tree. We can easily see that after 60 is inserted, the tree is still a binary search tree. Other than creating a node and attaching it to the tree, the process is identical to the search process and takes O(log *n*) time for a height-balanced complete binary search tree. Note that if the tree is complete before the insertion, it is probably not complete after the insertion. If the tree "looks like" a linked list, it is likely that the insertion will take O(*n*) running time. Once again, it is very desirable that a binary search tree is height-balanced.

Figure 15.17 Inserting 60 in the Binary Search Tree of Figure 15.13

Description

Figure 15.18 shows a sequence of insertions, starting from an empty binary search tree: 50, 20, 36, 66, 25, 5, 17, 84, and 44 are successively inserted.

Figure 15.18 A Sequence of Insertions in a Binary Search Tree

Description

15.5.3 Deleting from a Binary Search Tree

Deleting a value from a binary search tree can be more complex; indeed, we could be deleting a value in the middle of the tree. In order to delete a value, that value must be in the tree to begin with. Thus, we start by searching for the value. Assuming we find it, we have three different scenarios:

- The value is stored in a leaf,
- The value is stored in a node that has only one child, or
- The value is stored in a node that has two children.

Once again, we want the binary search tree to still be a binary search tree after the deletion. Also, we want the delete operation to be efficient, i.e., O(log *n*) for a height-balanced tree.

If the value we are deleting is stored in a leaf, we simply delete that leaf from the tree, i.e., the leaf's parent's child becomes *null.* **Figure 15.19** shows the resulting tree if we delete 25 from the tree in Figure 15.13. Obviously, it is still a binary search tree.

Figure 15.19 The Resulting Tree After Deleting 25 from the Tree in Figure 15.13

Description

If the value we are deleting is stored in a node that has only one child, the child of that node takes its place. Note that it could be either the left child or the right child. **Figure 15.20** shows the resulting tree if we delete 5 from the tree in Figure 15.13. Again, the resulting tree is a binary search tree. Note also that there could be a whole subtree underneath that child. If there is a whole subtree below the node containing 17, that whole subtree is a binary search tree itself, and although it is to the right of 5, it is to the left of 20; thus, every element in that subtree is smaller than 20. After the deletion takes place, that subtree is still to the left of 20.

Figure 15.20 The Resulting Tree After Deleting 5 from the Tree in Figure 15.13

Description

A more complex situation arises when we delete a value, for example 50, stored in a node that has two children. The general strategy is to replace that value with an existing value below it so that we keep the binary search tree property. We have two possible choices:

- Replace it with the largest value in the left subtree underneath the node containing the value we are deleting (for the tree in Figure 15.13, that is 44)
- Replace it with the smallest value in the right subtree underneath the node containing the value we are deleting (for the tree in Figure 15.13, that is 66)

Figure 15.21 shows the two such values of the tree in Figure

15.13 when we delete 50. The largest value on the left of 50 is 44, and the smallest value on the right of 50 is 66.

Figure 15.21 The Two Candidate Replacement Values When We Delete 50 from the Binary Search Tree in Figure 15.13

Description

The largest value in the left subtree is larger than all the other values in the left subtree and smaller than all the values in the right subtree (because it is in the left subtree, it is smaller than the value we are deleting, and therefore smaller than all the values in the right subtree).

The smallest value in the right subtree is smaller than all the other values in the right subtree and larger than all the values in the left subtree (because it is in the right subtree, it is larger than the value we are deleting, and therefore larger than all the values in the left subtree).

In order to find the largest value in the left subtree, we go left once and then go right as far as we can. **Figure 15.22** shows the path to find such value. The node containing 44 has no right child. In order to find the smallest value in the right subtree, we go right once and then go left as far as we can. **Figure 15.23** shows the path to find such value. The node containing 66 has no left child.

Figure 15.22 Finding the Largest Value (44) Left of 50 in the Tree of Figure 15.13

Description

Figure 15.23 Finding the Smallest Value (66) Right of 50 in the Tree of Figure 15.13

Description

When we write the code to delete a value, we simply pick one of the two choices and implement it. We do not implement both choices.

The value that is selected could be inside a leaf or inside a node that has one child. It is not possible that such a node has two children (otherwise, the value would not be the largest on the left side or the smallest on the right side; we would need to continue searching for such a value). Note that if it is inside a node that has one child, there could be a whole subtree underneath that child.

Either way, after we have placed that value inside the node containing the value that we are deleting, we simply delete that value from the tree. After the deletion, the resulting tree is a binary search tree. **Figures 15.24** and **15.25** show the resulting tree if we delete 50 from the tree in Figure 15.13. Figure 15.24 is based on the first scenario, and Figure 15.25 is based on the second scenario.

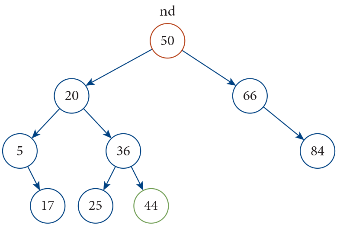
Figure 15.24 Using the First Scenario, the Resulting Tree After Deleting 50 from the Tree in Figure 15.13

Description

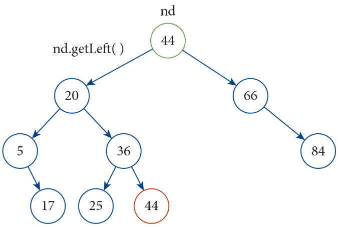
Figure 15.25 Using the Second Scenario, the Resulting Tree After Deleting 50 from the Tree in Figure 15.13

Description

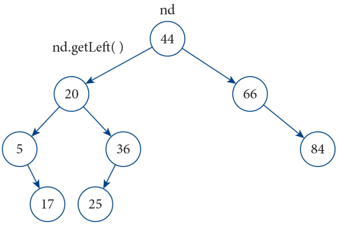
If the binary search tree is height-balanced, *delete,* like *search* and *insert,* takes O(log *n*). We first go down the tree in order to find the value to delete, then keep going down the tree to find its replacement value, and finally replace the deleted value and reset some links. Overall, we take a path from the root to either a leaf or a node with one child, “measuring” the height of the tree or less. Like *search* and *insert,* if the tree “looks like” a linked list, *delete* can take O(*n*).


nd
50
20
66
5
36
84
17
25
44

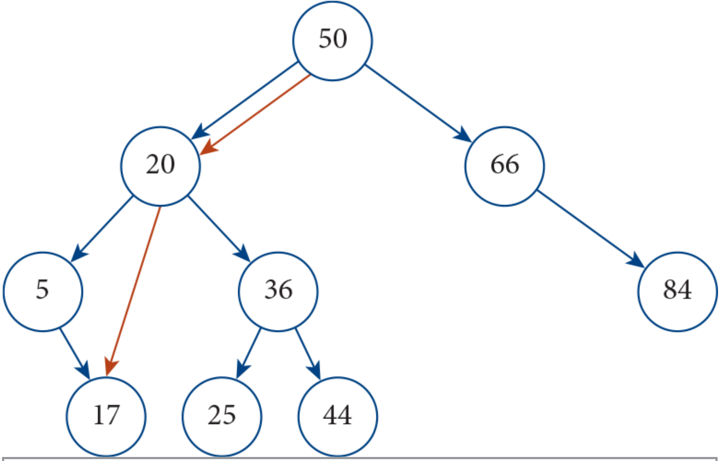
44 is the largest value
on the left side


nd
44
nd.getLeft( )
20
66
5
36
84
17
25
44

50 is changed to 44


nd
44
nd.getLeft( )
20
66
5
36
84
17
25

44 is deleted in the subtree
whose root contains 20

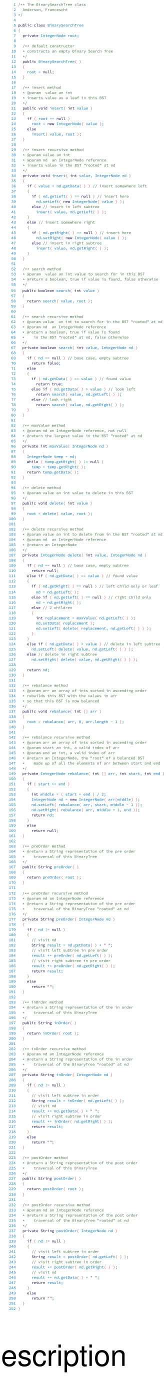

50
20
66
5
36
84
17
25
44

```
Pre order: 50 20 5 17 36 25 44 66 84
In order: 5 17 20 25 36 44 50 66 84
Post order: 17 5 25 44 36 20 84 66 50

Deleting 50
Pre order: 44 20 5 17 36 25 66 84
In order: 5 17 20 25 36 44 66 84
Post order: 17 5 25 36 20 84 66 44
...
Deleting 8
Pre order:
In order:
Post order:
```

15.6 Binary Search Tree Implementation

Again, to keep things simple and focus on the functionality of a binary search tree, we store integers in *IntegerNode* objects. We use the same *IntegerNode* class as the one in Example 15.3. **Example 15.6** shows the *BinarySearchTree* class.

Example 15.6 The *BinarySearchTree* Class



```java
/** The BinarySearchTree class
  Anderson, Franceschi
*/

public class BinarySearchTree
{
  private IntegerNode root;

  /** default constructor
  * constructs an empty Binary Search Tree
  */
  public BinarySearchTree( )
  {
    root = null;
  }

  /** insert method
  * @param  value an int
  * inserts value as a leaf in this BST
  */
  public void insert( int value )
  {
    if ( root == null )
      root = new IntegerNode( value );
    else
      insert( value, root );
  }

  /** insert recursive method
  * @param value an int
  * @param nd  an IntegerNode reference
  * inserts value in the BST "rooted" at nd
  */
  private void insert( int value, IntegerNode nd )
  {
    if ( value < nd.getData( ) ) // insert somewhere left
    {
      if ( nd.getLeft( ) == null ) // insert here
        nd.setLeft( new IntegerNode( value ) );
      else // insert in left subtree
        insert( value, nd.getLeft( ) );
    }
    else // insert somewhere right
    {
      if ( nd.getRight( ) == null ) // insert here
        nd.setRight( new IntegerNode( value ) );
      else // insert in right subtree
        insert( value, nd.getRight( ) );
    }
  }

  /** search method
  * @param  value an int value to search for in this BST
  * @return a boolean, true if value is found, false otherwise
  */
  public boolean search( int value )
  {
    return search( value, root );
  }

  /** search recursive method
  * @param value  an int to search for in the BST "rooted" at nd
  * @param nd  an IntegerNode reference
  * @return a boolean, true if value is found
  *    in the BST "rooted" at nd, false otherwise
  */
  private boolean search( int value, IntegerNode nd )
  {
    if ( nd == null ) // base case, empty subtree
      return false;
    else
    {
      if ( nd.getData( ) == value ) // found value
        return true;
      else if ( nd.getData( ) > value ) // look left
        return search( value, nd.getLeft( ) );
      else // look right
        return search( value, nd.getRight( ) );
    }
  }

  /** maxValue method
  * @param nd an IntegerNode reference, not null
  * @return the largest value in the BST "rooted" at nd
  */
  private int maxValue( IntegerNode nd )
  {
    IntegerNode temp = nd;
    while ( temp.getRight( ) != null )
      temp = temp.getRight( );
    return temp.getData( );
  }

  /** delete method
  * @param value an int value to delete in this BST
  */
  public void delete( int value )
  {
    root = delete( value, root );
  }

  /** delete recursive method
  * @param value an int to delete from in the BST "rooted" at nd
  * @param nd  an IntegerNode reference
  * @return an IntegerNode
  */
  private IntegerNode delete( int value, IntegerNode nd )
  {
    if ( nd == null ) // base case, empty subtree
      return null;
    else if ( nd.getData( ) == value ) // found value
    {
      if ( nd.getRight( ) == null ) // left child only or leaf
        nd = nd.getLeft( );
      else if ( nd.getLeft( ) == null ) // right child only
        nd = nd.getRight( );
      else // 2 children
      {
        int replacement = maxValue( nd.getLeft( ) );
        nd.setData( replacement );
        nd.setLeft( delete( replacement, nd.getLeft( ) ) );
      }
    }
    else if ( nd.getData( ) > value ) // delete in left subtree
      nd.setLeft( delete( value, nd.getLeft( ) ) );
    else // delete in right subtree
      nd.setRight( delete( value, nd.getRight( ) ) );

    return nd;
  }

  /** rebalance method
  * @param arr an array of ints sorted in ascending order
  * rebuilds this BST with the values in arr
  * so that this BST is now balanced
  */
  public void rebalance( int [] arr )
  {
    root = rebalance( arr, 0, arr.length - 1 );
  }

  /** rebalance recursive method
  * @param arr an array of ints sorted in ascending order
  * @param start an int, a valid index of arr
  * @param end an int, a valid index of arr
  * @return an IntegerNode, the "root" of a balanced BST
  *    made up of all the elements of arr between start and end
  */
  private IntegerNode rebalance( int [] arr, int start, int end )
  {
    if ( start <= end )
    {
      int middle = ( start + end ) / 2;
      IntegerNode nd = new IntegerNode( arr[middle] );
      nd.setLeft( rebalance( arr, start, middle - 1 ));
      nd.setRight( rebalance( arr, middle + 1, end ));
      return nd;
    }
    else
      return null;
  }

  /** preOrder method
  * @return a String representation of the pre order
  *    traversal of this BinaryTree
  */
  public String preOrder( )
  {
    return preOrder( root );
  }

  /** preOrder recursive method
  * @param nd an IntegerNode reference
  * @return a String representation of the pre order
  *    traversal of the BinaryTree "rooted" at nd
  */
  private String preOrder( IntegerNode nd )
  {
    if ( nd != null )
    {
      // visit nd
      String result = nd.getData( ) + " ";
      // visit left subtree in pre order
      result += preOrder( nd.getLeft( ) );
      // visit right subtree in pre order
      result += preOrder( nd.getRight( ) );
      return result;
    }
    else
      return "";
  }

  /** inOrder method
  * @return a String representation of the in order
  *    traversal of this BinaryTree
  */
  public String inOrder( )
  {
    return inOrder( root );
  }

  /** inOrder recursive method
  * @param nd an IntegerNode reference
  * @return a String representation of the in order
  *    traversal of the BinaryTree "rooted" at nd
  */
  private String inOrder( IntegerNode nd )
  {
    if ( nd != null )
    {
      // visit left subtree in order
      String result = inOrder( nd.getLeft( ) );
      // visit nd
      result += nd.getData( ) + " ";
      // visit right subtree in order
      result += inOrder( nd.getRight( ) );
      return result;
    }
    else
      return "";
  }

  /** postOrder method
  * @return a String representation of the post order
  *    traversal of this BinaryTree
  */
  public String postOrder( )
  {
    return postOrder( root );
  }

  /** postOrder recursive method
  * @param nd an IntegerNode reference
  * @return a String representation of the post order
  *    traversal of the BinaryTree "rooted" at nd
  */
  private String postOrder( IntegerNode nd )
  {
    if ( nd != null )
    {
      // visit left subtree in order
      String result = postOrder( nd.getLeft( ) );
      // visit right subtree in order
      result += postOrder( nd.getRight( ) );
      // visit nd
      result += nd.getData( ) + " ";
      return result;
    }
    else
      return "";
  }
}
```

Description

Like we did for the traversal methods in the *BinaryTree* class, we use recursion to code the *insert* method. The *private* recursive method is at lines 29–50 and the *public* wrapper method is at lines 17–27. The recursive method inserts *value,* an *int,* its first parameter, in a binary search tree rooted at *nd,* an *IntegerNode,* its second parameter (line 34). At line 36, we test if *value* is smaller than the value stored inside *nd*. If it is, we insert in the left subtree of nd. If *nd* has no left child (line 38), we have found the point of insertion. We instantiate a node containing *value* and set it as the left child of *nd* (line 39). If *nd* has a left child, we make a recursive call to insert in the left subtree of *nd* (line 41). If *value* is larger than the value stored inside *nd* (line 43), we insert as the right child of *nd* if *nd* does not have a right child (lines 45–46), or we insert in the right subtree of *nd* if *nd* has a right child by making a recursive call (line 48).

The wrapper method instantiates *root* with *value* (line 24) if the tree is empty (line 23). Otherwise, it calls the recursive method to insert *value* in the subtree rooted at *root,* i.e., the whole tree (line 26).

We also use recursion to code the *search* method. The *private* recursive method is at lines 61–80 and the *public* wrapper method is at lines 52–59. The recursive method searches for *value,* an *int,* in a subtree rooted at a node *nd,* both parameters of the method (line 67). The wrapper method searches for *value* in the subtree rooted at *root,* i.e., the whole tree. Both methods return *true* if we find *value, false* if we do not. If *nd* is *null* (line 69), then the subtree rooted at *nd* is empty and we will not find *value* inside it; we return *false* at line 70. If *nd* is not *null,* we test if it contains *value* at line 73 and return *true* in that case (line 74). Otherwise, we compare *value* to the value inside *nd* in order to determine whether we search the left subtree (line 76) or the right subtree (line 78) and return whatever the recusive call to the *search* method returns.

At lines 163–251, we include methods to traverse the tree in pre order, in order, and in post order; they are identical to the ones in Example 15.4. These methods are very useful to check that the resulting tree is correct after calling the *insert* and *delete* methods in a client class.

We also use recursion to delete a value from the tree. The *private* recursive method (lines 102–130) deletes *value,* its first parameter, from the subtree rooted at node *nd,* its second parameter (line 107). It returns an *IntegerNode* because we want to use the returned value as the left or right child of a node, i.e., use it as the argument passed to either the *setLeft* or *setRight* methods (lines 121, 125, and 127).

At line 111, we test if the current node, *nd*, contains the value we are looking to delete. If it does, there are three scenarios: *nd* does not have a right child, i.e., is either a leaf or has a left child only (line 113), has a right child only (line 115), or has two children (line 117). If it is a leaf or only has a left child, we simply assign that left child to *nd* (line 114). That converts a grandparent-grandchild relationship between the parent of *nd* and the left child of *nd* to a parent-child relationship; note that the left child of *nd* could be *null*. We do something similar if *nd* has only a right child (line 116). If *nd* has two children, we first call the *maxValue* method in order to find the largest value, *replacement,* in the left subtree of *nd* (line 119); *maxValue,* a *private* utility method, is coded at lines 82–92. Starting at its node parameter *nd*, it finds and returns the largest value within the tree rooted at *nd*. Note that when that method is called, *nd* is guaranteed to have two children. Thus, *temp,* which is assigned *nd.getLeft( )* at line 88, is guaranteed not to be *null,* and there is no risk of a *NullPointerException* at line 89. The value returned by *maxValue* is the value we want to place inside *nd*; we do that at line 120. Then, we make a recursive call to the *delete* method in order to delete that replacement value in the tree rooted at the left child of *nd* (line 121). As part of the same statement making the recursive call (line 121), we also attach the *IntegerNode* returned by the method as the left child of *nd*. Most of the time, that attachment will not change anything in the tree as we attach *nd* to its child. Eventually, a base case is reached, and the deleting actually takes place. Note that because the replacement value is the largest value in the left subtree under *nd,* it is guaranteed not to have a right child (it may or may not have a left child). Thus, when we call the *delete* method to delete, we are guaranteed to fall either in the “delete a leaf” or “delete a node with one child” scenario. These steps are illustrated in **Figure 15.26**.

Figure 15.26 Steps in Deleting a Value (50) Stored in a Node with Two Children

Description

At line 124, we test if the value inside *nd* is greater than *value.* If it is, we make a recursive call to delete *value* within the subtree rooted at the left child of *nd* (line 125). We also set the node returned by the recursive call to the *delete* method to be the left child of *nd*. The process is similar if the value inside *nd* is smaller than *value* (lines 126–127). Again, most of the time, that attachment will not change anything in the tree as we attach *nd* to its child. Eventually, a base case is reached, and the deleting actually takes place.

At line 129, we return *nd*. **Figure 15.27** shows the links that are being reset (in red) when we call *delete* with argument 5.

Figure 15.27 In Red, the Links Reset When Deleting 5

Description

Table 15.4 lists what happens when we call the *delete* method with argument 5. The notation "node*(x)"* represents the node that contains the value *x.* As recursive calls are made, the first argument of *delete* is always 5, while its second argument starts with *root* and goes through every node in the path between *root* and the node containing 5 (node(5)) as the method searches for and finds that node. When *delete* is called (call # 1) with argument node(50), line 125 executes and eventually the method returns node(50) at line 129. The argument passed to *setLeft* at line 125 is the node returned by the call to *delete* with argument node(20), the left child of node(50). When *delete* is called (call # 2) with argument node(20), line 125 executes again and eventually the method returns node(20). Thus, node(20) is the argument passed to *setLeft* above; that does nothing since the left child of node(50) is already node(20). During call # 2, we set the left child of node(20) to be the node returned by *delete* when called with argument node(5). When *delete* is called (call # 3) with argument node(5), lines 116 and 120 evaluate to *true,* and line 121 executes. It changes the value of *nd* to be node(17), which is then returned at line 129. Thus, the left child of node(20) becomes node(17) and node(5) is no longer reachable, and therefore marked for garbage collection. Tracing what happens when we delete a value stored in a leaf, for example 84, or a value stored in a node that has two children, for example 50, are left as exercises.

The Recursive Calls of the *delete* Method When Deleting 5 (Node Has One Child)

"Method call"	Argument (nd)	Returned Node	Setting Children	Line
1: delete(5, root)	node(50)	node(50)	node(50).setLeft(node(20))	125
2: delete(5, node(20))	node(20)	node(20)	node(20).setLeft(node(17))	125
3: delete(5, node(5))	node(5)	node(17)		

Description

Example 15.7 shows how to build the tree in Figure 15.13, looping through an array of values and calling the *insert* method (lines 10–12). At lines 14–16, we call the three traversal methods to check that the tree is the one in Figure 15.13. At line 18, we declare an array of values to be deleted from the tree, including *9* and *8*, two values that are not in the tree. At lines 19–29, we loop through the array, testing many situations, including a value (*9*) not in the tree and trying to delete (*8*) from an empty tree.

Example 15.7 The *BinarySearchTreeClient* Class

```
 1 /** The BinarySearchTreeClient class
 2   Anderson, Franceschi
 3 */
 4
 5 public class BinarySearchTreeClient
 6 {
 7   public static void main( String [] args )
 8   {
 9     BinarySearchTree bst = new BinarySearchTree( );
10     int [] insertions = { 50, 20, 36, 66, 25, 5, 17, 84, 44 };
11     for( int value : insertions )
12       bst.insert( value );
13
14     System.out.println( "Pre order: " + bst.preOrder( ) );
15     System.out.println( "In order: " + bst.inOrder( ) );
16     System.out.println( "Post order: " + bst.postOrder( ) );
17
18     int [] deletions = { 50, 20, 36, 66, 9, 84, 17, 25, 5, 44, 8 };
19     for( int value : deletions )
20     {
21       System.out.println( "\nDeleting " + value );
22       bst.delete( value );
23       System.out.println( "Pre order: " + bst.preOrder( ) );
24       System.out.println( "In order: " + bst.inOrder( ) );
25       System.out.println( "Post order: " + bst.postOrder( ) );
26     }
27   }
28 }
```

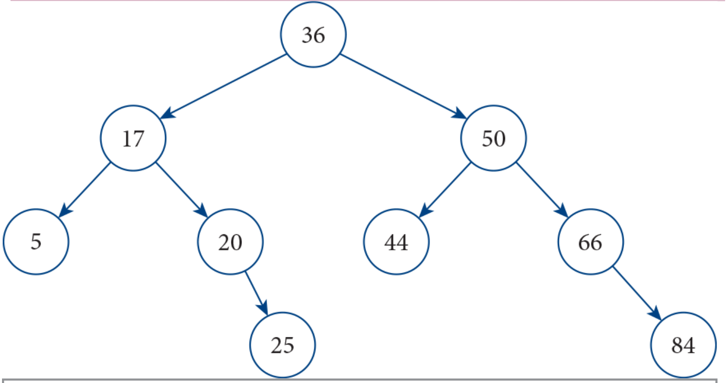
Description

At lines 23–25, we call the three traversal methods after each deletion to check the correctness of the deletion. The output is shown in **Figure 15.28**.

Figure 15.28 The Output of Example 15.7

Description

Index	0	1	2	3	4	5	6	7	8
Value	5	17	20	25	36	44	50	66	84



```
Pre order: 5 17 20 25 36 44 50 66 84
In order: 5 17 20 25 36 44 50 66 84
Post order: 84 66 50 44 36 25 20 17 5

After rebuilding the tree
Pre order: 36 17 5 20 25 50 44 66 84
In order: 5 17 20 25 36 44 50 66 84
Post order: 5 25 20 17 44 84 66 50 36
```

15.7 Rebalancing a Binary Search Tree

We have seen that if a binary search tree is height-balanced, *search, insert,* and *delete* are O(log *n*) and therefore very efficient. However, after many *insert* operations, it is very likely that the binary search tree is no longer height-balanced. In fact, it could degenerate into something that looks like a linked list. Inserting successively 1, 2, 3, 4, and 5 results in the binary search tree in Figure 15.16.

We can rebalance any binary search tree so that it is either complete or almost complete. We do this in two steps:

- Traverse the binary search tree in order and unload it into an array (the array will be automatically sorted in ascending order).
- From the sorted array, build a binary search tree that is height-balanced.

For the first step, if we keep track of the number of elements inserted in the tree, we can size the array appropriately. If we do not, we can build an *ArrayList* as we traverse the tree in order, then unload the *ArrayList* elements into an array. This part is left as an exercise. We focus on how to rebuild a balanced binary search tree from a sorted array.

If we traverse in order and unload the binary search tree in Figure 15.13 into an array, we get the array shown in **Figure 15.29**.

Figure 15.29 The Array Resulting from Traversing In Order the Tree in Figure 15.13

Description

Our algorithm to build a balanced binary search tree is as follows:

- Define *root* as holding the value in the middle of the array, here 36.
- Construct a height-balanced binary search tree with the array elements on the left half of the array (5, 17, 20, 25) and attach its root as the left child of *root.*
- Construct a height-balanced binary search tree with the array elements on the right half of the array (44, 50, 66, 84) and attach its root as the right child of *root.*

This algorithm is recursive and steps 2 and 3 are recursive calls, building smaller balanced binary search trees. Let’s apply the algorithm to the array in Figure 15.29. The root of the tree contains 36, the middle element. Its left child contains the middle element of the left half of the array, 17(0+3)/2 = 1 is the middle index of the left half of the array). Its right child contains the middle element of the right half of the array, 50((5+8)/2 = 6 is the middle index of the right half of the array). The left child of the node containing 17 contains 5, the only element left on the left side of the array. The right child of the node containing 17 contains the middle element of the slice of the array that contains 20 and 25, i.e., 20((2+3)/2 = 2). As the algorithm ends, we get the tree shown in **Figure 15.30**. Every level is completely filled except the last level. We can see that the tree is height-balanced.

Figure 15.30 The Balanced Binary Search Tree Built from the Array in Figure 15.29

Description

As for our three traversals, *insert, search,* and *delete* methods, we use a *public* wrapper method and a *private* recursive method to build a balanced binary search tree from a sorted array. Our recursive *rebalance* method builds a balanced binary search tree from a slice of a sorted array. We can define a slice of an array with three things: the array itself, the starting index, and the ending index. Thus, our recursive method accepts those three as parameters (line 149). It returns the root of the tree that it is building, so that it can be attached as the left or right child of another root from a bigger tree as we make the recursive calls. In the general case, it does the following:

- Create a node (*nd*) with the middle element of the array slice (lines 153–154); that node is the root of the subtree that the method is building,
- Make a recursive call to build a balanced binary search tree with the left half of the subarray (line 155). That recursive call returns a node, the root of the subtree that the method is building; attach that node as the left child of *nd* (line 155), and
- Make another recursive call to build a balanced binary search tree with the right half of the subarray (line 156). That recursive call returns a node, the root of the subtree that the method is building; attach that node as the right child of *nd* (line 156).

If the array slice is empty, the method "builds" an empty tree and returns *null* (lines 159–160); that is the base case.

Our *public* wrapper method only needs to accept one parameter, an array of *ints* (line 137). It calls the recursive method to build a balanced binary search tree for the array slice whose starting index is 0 and ending index is the last array index (line 139), i.e., the whole array.

Example 15.8 shows how to use the *rebalance* method of Example 15.6 to rebuild a height-balanced binary search tree. We first build a binary search tree that looks like a linked list, inserting successively all the values of a sorted array (lines 10–12). At line 18, we rebuild the tree so that it is a height-balanced binary search tree. At lines 21–23, we call the three traversal methods to check that the tree is the one in Figure 15.30. The output is shown in **Figure 15.31**.

Example 15.8 The *BinarySearchTreeClient2* Class

```
 1 /** The BinarySearchTreeClient2 class
 2   Anderson, Franceschi
 3 */
 4
 5 public class BinarySearchTreeClient2
 6 {
 7   public static void main( String [] args )
 8   {
 9     BinarySearchTree bst = new BinarySearchTree( );
10     int [] insertions = { 5, 17, 20, 25, 36, 44, 50, 66, 84 };
11     for( int value : insertions )
12       bst.insert( value );
13
14     System.out.println( "Pre order: " + bst.preOrder( ) );
15     System.out.println( "In order: " + bst.inOrder( ) );
16     System.out.println( "Post order: " + bst.postOrder( ) );
17
18     bst.rebalance( insertions );
19
20     System.out.println( "\nAfter rebuilding the tree" );
21     System.out.println( "Pre order: " + bst.preOrder( ) );
22     System.out.println( "In order: " + bst.inOrder( ) );
23     System.out.println( "Post order: " + bst.postOrder( ) );
24   }
25 }
```

Description

Figure 15.31 The Output of Example 15.8

Description

15.8 Programming Activity 2: Working with Binary Search Trees

In this activity, you will simulate and test the running time of the *insert* method. In order to do this, you will modify the *insert* method of the *BinarySearchTree* class and keep track of the number of steps it takes to insert.

Instructions

Copy the source files in the Programming Activity 2 folder for this chapter to a folder on your computer. Note that all files should be in the same folder.

Inside the *BinarySearchTree* class, you need to modify and code the following methods (the binary tree stores integer values):

Modify the *insert* method so that it keeps track of the number of steps it took to insert (the number of steps is defined as the number of edges from the root to the newly inserted node). You can code the *insert* method so that it returns the number of steps. Alternatively, you can use a *static* instance variable in the class and update it; if you use a *static* instance variable, you need to provide methods to access it from outside the class and reset it to 0.

Code a method that returns an array of integers representing all the values, in ascending order, stored in the binary search tree.

Inside a client class, build a binary search tree storing integers as follows:

- Generate 100000 random integers with values from 0 to 1000000 (to keep it simple, duplicates are accepted) and insert them all in the binary search tree.
- Then, generate 1000 random integers with values from 0 to 1000000 (again, duplicates are accepted) and insert them in the binary search tree, and calculate and output the

average number of steps it took to insert a value.

- Restore the binary search tree to a height-balanced state. Then, generate 1000 random integers with values from 0 to 1000000 (again, duplicates are accepted) and insert them in the binary search tree, and calculate and output the average number of steps it took to insert a value.
- The values 1000, 100000, and 1000000 should be constants (and you should use those constants).

On a separate Word document (1 page max), report your results (first average number of steps to insert, and second average number of steps to insert). In both cases, how does the average number of steps relate to the number of elements in the binary search tree?

Troubleshooting

- Code one method at a time and test your results.
- Check that your results correlate with the math computations in this chapter.

DISCUSSION QUESTIONS

1. **How do your results compare to the math calculations in this chapter?**
2. **Did it help with the running time of insert to reconstruct a height-balanced binary search tree? How much?**

© Lois GoBe/Shutterstock

CHAPTER REVIEW

15.9 Chapter Summary

- A tree is a data structure simulating the hierarchical nature of a tree, including a root and its descendants.
- Nodes in a tree are connected using parent-child relationships.
- The degree of a node is its number of children.
- Sibling nodes are nodes that share the same parent node.
- A leaf is a node that has no child.
- The depth of a node is the number of edges on the path from the root to that node.
- The level of a node is also the number of edges on the path from the root to that node.
- The height of a node is the largest number of edges on the path from the node to a leaf.
- The height of a tree is the height of its root.
- A binary tree is a tree where all the nodes have degree 0, 1, or 2.
- We can traverse a tree in level order, pre order, in order, or post order.
- Traversing the tree in level order, we list the elements level by level, left to right for each level.
- Traversing in pre order, in order, and post order involve

walking around the tree counterclockwise.

- Traversing in pre order, we list each element the first time we see it.
- Traversing in order, we list each element the second time we see it.
- Traversing in post order, we list each element the last time we see it.
- Pre order, in order, and post order traversals are easy to implement using recursion.
- A complete binary tree is a binary tree where every level is full except the last one; the last level is filled left to right.
- A full binary tree is a complete binary tree where the last level is also full.
- In a full binary tree of height h, we have $n = 2^h - 1$ elements.
- A binary search tree is a binary tree such that:
    - All the elements to the left of the root have a value smaller than the root's value.
    - All the elements to the right of the root have a value larger than the root's value.
    - The left subtree of a binary search tree is a binary search tree itself.
    - The right subtree of a binary search tree is a binary search tree itself.
- An empty binary tree is a binary search tree.
- If we traverse a binary search tree in order, the values come out in ascending order.
- In a binary search tree, we insert as a leaf and maintain the binary search tree property.
- In a binary search tree, we maintain the binary search tree property when we delete.
- If a binary search tree is height-balanced, *insert, delete,* and *search* take O(log n) running time, where n is the

number of values stored in the tree.

- We can build a height-balanced binary search tree from an array sorted in ascending order.

15.10 Exercises, Problems, and Projects

15.10.1 Multiple Choice Exercises

1. The node at the “top” of the tree hierarchy is called the
    - head
    - tail
    - root
5. The degree of a node is
    - its number of parents
    - its number of children
    - its number of siblings
    - its stored value
10. A leaf is a node of degree
    - −1
    - 0
    - 1
    - 2
15. The level of a node and the depth of a node have equal values
    - True
    - False
18. The height of a tree is equal to
    - the depth of its left most leaf
    - the depth of its root

- the number of children of its root
- the height of its root

23. The pre order, in order, and post order traversals involve walking around the tree

    - level by level
    - clockwise
    - counterclockwise

27. In a binary tree, every node has

    - no child
    - exactly 1 child
    - 0, 1, or 2 children
    - 1 or 2 children

32. When traversing a binary tree in order, we list each node’s element

    - the first time we see it
    - the second time we see it
    - the third time we see it
    - the fourth time we see it

37. When traversing a binary search tree in order, the elements come out

    - in descending order
    - in no particular order
    - in ascending order

41. If a binary search tree is complete, the worst-case running time of *insert, search,* and *delete* is

    - O(l)
    - O(log *n*)

- O(*n*)
- O(*n* log *n*)

46. If we loop through all the elements of a sorted array from the first to the last index, and insert the elements in a binary search tree in that order, we will get

    - A balanced binary search tree
    - A complete binary search tree
    - A full binary search tree
    - A binary search tree that looks like a linked list

51. Check all that apply: We want to build a balanced binary search tree from an array sorted in ascending order.

    - The middle array element will be stored in the root.
    - The left half of the array (minus the middle element) will make up the left subtree.
    - The right half of the array (minus the middle element) will make up the right subtree.
    - The balanced binary search tree will always be full.

15.10.2 Reading and Understanding Code

1. Within the binary tree of Figure 15.3, the *IntegerNode nd* is a reference to the node containing 33:

   Redraw the tree after the following code is executed:

   ```
   nd.setRight( nd.getRight( ).getRight( ) );
   ```

   Description

2. Within the binary tree of Figure 15.3, the *IntegerNode nd* is a reference to the node containing 33:

   Redraw the tree after the following code is executed:

   ```
   IntegerNode temp = nd.getLeft( ).getRight( );
   nd.getLeft( ).setRight( temp.getRight( ) );
   ```

Description

3. The pre order and in order traversals of a binary tree are as follows:

```
In order:   55 17 74 100 54 22 30  8 12
Pre order: 100 17 55  74 22 54  8 30 12
```

Description

What value is stored inside the root?
What values are stored in the left subtree?
What values are stored in the right subtree?

4. The traversals of a binary tree are as follows:

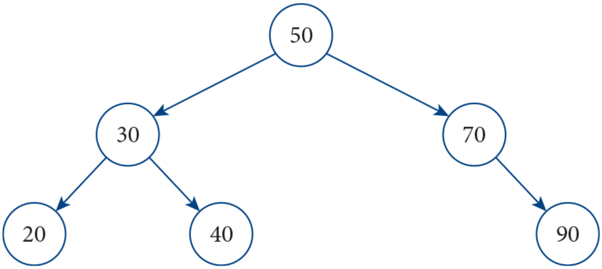
```
Pre order:  88 44 60 25  10 33  40 100
In order:   60 44 10 25  88 33 100  40
Post order: 60 10 25 44 100 40  33  88
```

Description

Draw the tree.

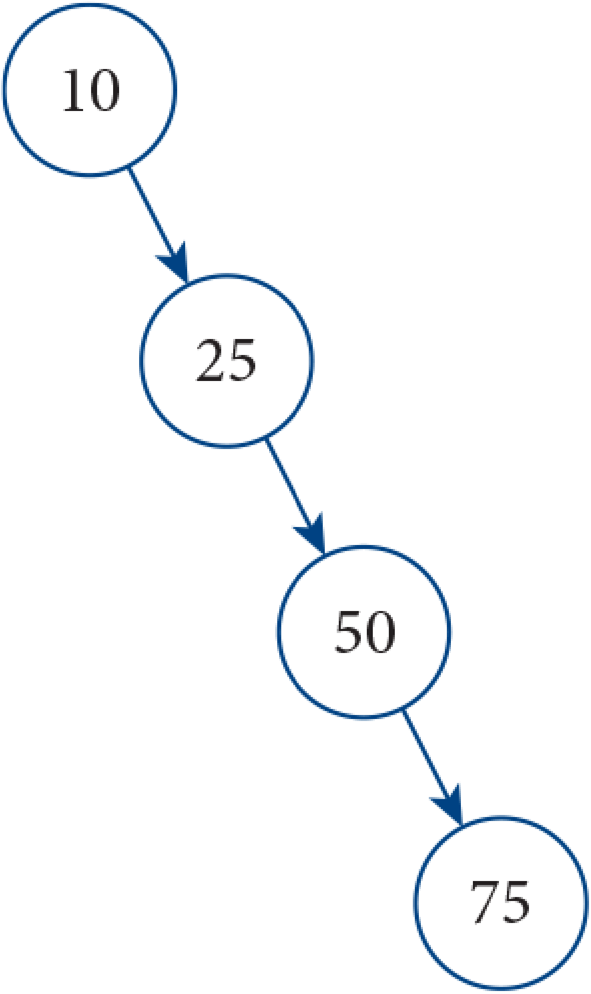
5. We successively insert 100, 125, 75, 87, 50, 150, 175, and 25 in a binary search tree; draw the tree.

6. What succession of inserts would result in the binary search tree below?



Description

7. What succession of inserts would result in the binary search tree below?



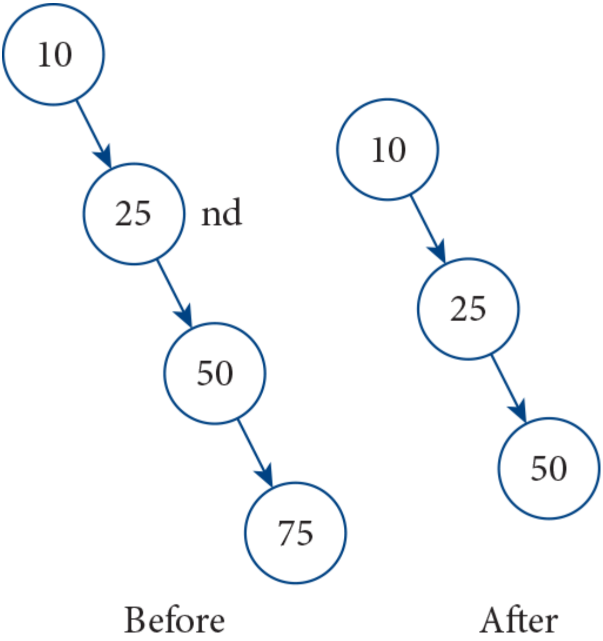
Description

8. Build a table similar to Table 15.4 when deleting 84 from the binary search tree in Figure 15.13.
9. Build a table similar to Table 15.4 when deleting 50 from the binary search tree in Figure 15.13.

15.10.3 Fill In the Code

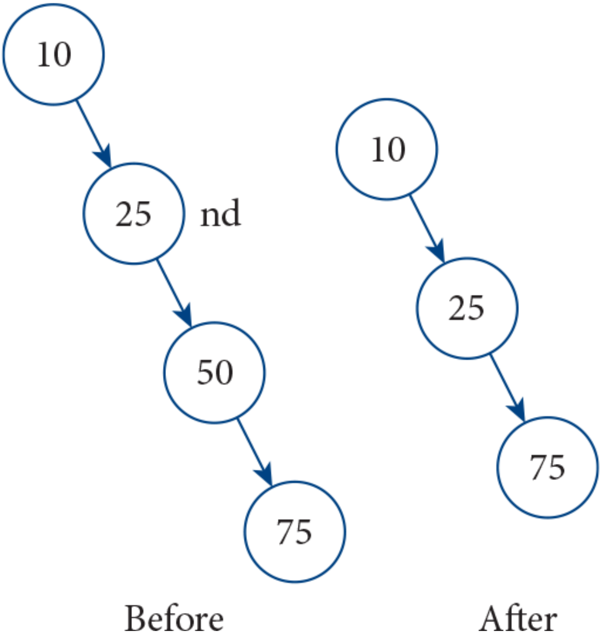
1. Assume that you are coding somewhere inside the

BinaryTree class and that *nd* is a reference to the node containing 25. Write the code to go from the Before to the After picture.



Description

2. Assume that you are coding somewhere inside the *BinaryTree* class and that *nd* is a reference to the node containing 25. Write the code to go from the Before to the After picture.



Description

3. This method of the *BinaryTree* class returns *true* if a binary tree is empty, *false* otherwise.

```
public boolean isEmpty( )
{
   // Your code goes here
}
```

Description

4. This method of the *BinaryTree* class returns *true* if a binary tree has exactly one element, *false* otherwise.

```
public boolean hasExactlyOne( )
{
   // Your code goes here
}
```

Description

5. This method of the *BinaryTree* class returns *true* if a binary tree has at least one element and if the root has 2 children, *false* otherwise.

```
public boolean rootHas2Children( )
{
  // Your code goes here

}
```

Description

6. This method of the *BinaryTree* class returns *true* if a binary tree stores at least three values, *false* otherwise.

```
public boolean storesAtLeast3Values( )
{

   // Your code goes here

 }
```

Description

7. The *public* method of the *BinaryTree* class returns the number of elements in a binary tree that are odd (assume that we store integer values in the binary tree).

```
public int numberOfOddValues( )
{
  // Your code goes here

}

private int numberOfOddValues( IntegerNode nd )
{
  // Your code goes here

}
```

Description

15.10.4 Identifying Errors in Code

1. Where could an error be generated with the following code? How would you fix it?

   ```
   public String preOrder( )
   {
     return preOrder( root );
   }
   private String preOrder( IntegerNode nd )
   {
     String result = nd.getData( ) + " ";
     result += preOrder( nd.getLeft( ) );
     result += preOrder( nd.getRight( ) );
     return result;
   }
   ```

   Description

2. Instead of the *attachLeft* method of the *BinaryTree* class of the chapter, we code this *attachLeft* method instead. What is the error?

   ```
   public void attachLeft( BinaryTree bt )
   {
     root = root.getLeft( bt.root );
   }
   ```

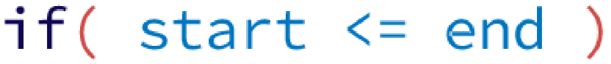
   Description

3. Instead of the *public delete* wrapper method of the *BinarySearchTree* class of the chapter, we code this *delete* method instead, which returns *true or false.* When we try to delete a value that is not found, it returns *true;* explain why.

   ```
   public boolean delete( int value )
   {
     return delete( value, root ) != null;
   }
   ```

   Description

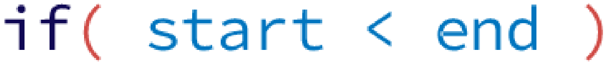
4. In the *rebalance* recursive method of the *BinarySearchTree* class of the chapter, we change:



```
if( start <= end )
```

Description

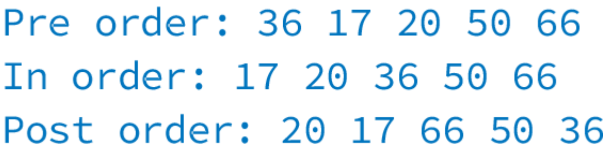
to



```
if( start < end )
```

Description

With the same client as in the chapter, the output is now:

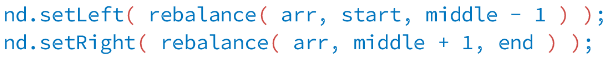


```
Pre order: 36 17 20 50 66
In order: 17 20 36 50 66
Post order: 20 17 66 50 36
```

Description

What happened to the missing numbers (5, 25, 44, 84)?

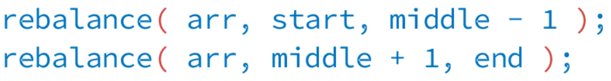
5. In the *rebalance* recursive method of the *BinarySearchTree* class of the chapter, we change:



```
nd.setLeft( rebalance( arr, start, middle - 1 ) );
nd.setRight( rebalance( arr, middle + 1, end ) );
```

Description

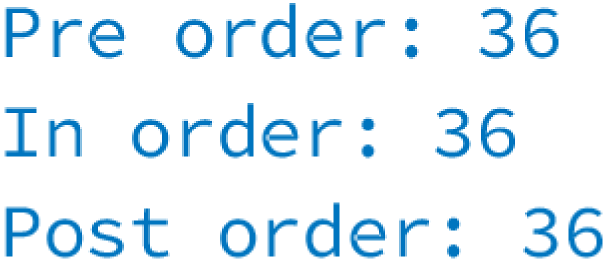
to



```
rebalance( arr, start, middle - 1 );
rebalance( arr, middle + 1, end );
```

Description

With the same client as in the chapter, the output is now:



```
Pre order: 36
In order: 36
Post order: 36
```

Description

It seems that there is only one value, 36, in the binary search tree. How did this happen?

15.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. In the *BinaryTree* class of the chapter, we add these methods:

```
public void preOrder2( )
{
  preOrder2( root );
}

private void preOrder2( IntegerNode nd )
{
  System.out.print( nd.getData( ) + " " );
  preOrder2( nd.getLeft( ) );
  preOrder2( nd.getRight( ) );
}
```

Description

When we call *preOrder2* from the client, we get a runtime exception as shown below; explain what the problem is (i.e., what is *null)* and how to fix it.

```
Exception in thread "main" java.lang.NullPointerException;
Cannot invoke "IntegerNode.getData()" because "<parameter1>" is null
   at BinaryTree.preOrder2(BinaryTree.java:125)
   ...
```

Description

2. In the *attachLeft* of the *BinaryTree* class of the chapter, we

change:

```
root.setLeft( bt.root );
```

Description

to

```
root.setLeft( bt );
```

Description

At compile time, we get the following error:

```
BinaryTree.java:24: error: incompatible types: BinaryTree
cannot be converted to IntegerNode
    root.setLeft( bt );
```

Description

Explain what the problem is and how to fix it.

3. Instead of the *public insert* wrapper method of the *BinarySearchTree* class of the chapter, we code this *insert* method instead.

```
public void insert( int value )
{
     insert( value, root );
}
```

Description

When we call it from the client, we get a runtime exception as shown below; explain what the problem is (i.e., what is *null)* and how to fix it.

```
Exception in thread "main" java.lang.NullPointerException;
Cannot invoke "IntegerNode.getData()" because "<parameter2>" is null
    at BinarySearchTree.insert(BinarySearchTree.java:32)
    at BinarySearchTree.insert(BinarySearchTree.java:26)
    at BinarySearchTreeClient.main(BinarySearchTreeClient.java:11)
```

Description

15.10.6 Write a Short Program

1. Inside the *BinaryTree* class, write a method that calculates the number of nodes that have exactly 1 child.
2. Inside the *BinaryTree* class, write a method that returns all the nodes in the binary tree that have exactly 2 children; you can return that list as an *ArrayList.*
3. Inside the *BinaryTree* class, write a method that returns the rightmost data stored in the binary tree; the method should throw an exception if the binary tree is empty.
4. Inside the *BinaryTree* class, write a method that returns all the values stored in the binary tree that are multiples of 10; you can return that list as an *ArrayList.*
5. Inside the *BinaryTree* class, write a method that returns *true* if the binary tree is a full binary tree, *false* otherwise.
6. Inside the *BinaryTree* class, write a method that returns *true* if the binary tree is a complete binary tree, *false* otherwise.
7. Inside the *BinaryTree* class, write a method that returns *true* if the binary tree is height-balanced, *false* otherwise.
8. Inside the *BinaryTree* class, write a method that returns the maximum sum of all the paths from the root to a leaf.
9. Inside the *BinarySearchTree* class, write a method that returns all the values stored in the binary search tree in ascending order; you can return that list as an *ArrayList.*
10. Inside the *BinarySearchTree* class, modify the *delete* method so that when we delete a node that has two children, we use the other scenario instead.

15.10.7 Programming Projects

1. Write a modified *BinaryTree* class so that we insert in such a way that the binary tree is always complete.

2. Same as 47 and also include a *delete* method so that we delete the last element that was inserted in the binary tree.
3. Same as 48 and also include another *delete* method so that we delete any element and replace it with the deepest/ rightmost ("last") element.
4. Write a modified *BinaryTree* class so that it includes a method to build a complete binary tree from a linked list.
5. Write a program that includes a *BinaryTree* class that stores *chars;* each *char* can be a digit (0 to 9) or an arithmetic operator (+ , –, *, /). In the client, build a binary tree such that, when traversing the tree in order, correct arithmetic expressions, such as 3 + 5 * 6, are returned.
6. Write a program that is the same as #51 but add parentheses around each operation (for example, an in order traversal would return

   $$((3 + 5) * 6), \text{or} (((6 * 7) - 8) + (9 * 4))$$

7. Write a program that includes a *BinaryTree* class that stores *Strings;* each *String* can be either AND, OR, a, b, or c. A binary tree stores a valid boolean expression (a and b are *boolean* variables). In the client, build a binary tree such that, when traversing the tree in order, correct boolean expressions, such as a AND b OR a, are returned.
8. Same as #53 but add parenthesis around each operation (for example, an in order traversal would return ((a AND b) OR b), or (((a OR b) AND b) AND (c OR a)).
9. Write a program that includes a *BinaryTree* class that stores *Strings;* each *String* is the name of a directory, for example bin, home, or / (the root in Linux). In the client, build a binary tree such that, when searching for a *String* (directory or file name) in the tree, the correct path is

returned (for example /bin/java, or /home/user/msmith). Include code in the client to build a tree of at least 10 elements and enable the user to search for a file path.

10. Write a program that includes a *BinaryTree* class that stores *chars;* each *char* is either a . (dot) or a - (hyphen). The binary tree stores letters coded using Morse code. The Morse code representation of a letter in the tree is its path from the root to that letter (for example, a is .-, b is - ..., c is -.-., d is -.., e is ., ...). Only consider letters from a to z. Add a method in the *BinaryTree* class that returns the Morse code of a letter. In the client, build a binary tree that stores all the letters from a to z. Ask the user to input a letter and return its Morse code representation by searching the tree.
11. Write a program that includes a *BinaryTree* class that stores pairs of *Strings;* that tree will be used as a family tree. The pair of *Strings* is meant to be one person of the family and his or her spouse (or *null* if that person is not married). Assume that people can only have two children. Each *String* is the name of a person, for example *Jill* or *Adam.* Add methods to do the following: based on somebody's name, return *true* if that person is in the tree and is married; based on one name, return the name of that person's sibling (it may have a *null* value); based on somebody's name, return the list of all the descendants. In the client, build a binary tree and test those methods based on user input. Include code in the client to build a tree of at least 10 elements.
12. Write a program that includes a *BinarySearchTree* class that stores information about employees. An *Employee* is defined as having an email, a job title, and a salary. Order the binary search tree based on the email. Add methods to

search based on the email and the job title. For these two methods, return an *Employee* object reference. Add methods to return a list of *Employees* who have a salary greater than or equal to a certain value and a list of *Employees* who have an email that ends with *edu.* In the client, build a binary search tree and test all those methods based on user input. Include code in the client to build a tree of at least 10 elements.

13. Same as #58 and include a method that returns an *ArrayList* of *Employees* sorted in ascending order based on email. In the *Employee* class, include two *static* methods: one that accepts an *ArrayList* of *Employees* and return it in ascending order based on salary; another one that accepts an *ArrayList* of *Employees* and return an array of *Employees.* In the *BinarySearchTree* class, also include a method to rebalance the tree based on salary ordering. In the client, build a binary search tree and test all those methods. Include code in the client to build a tree of at least 10 elements.
14. Write a program that includes a *BinarySearchTree* class that stores information about television channels. A *Channel* is defined as having a number, a description, and whether it is a cable channel or not. Order the binary search tree based on the channel number. Add a method to search based on the channel number and return a *Channel* object reference. Add a method that returns the first *Channel* object you find that matches a certain description. Add methods to return a list of *Channels* that are cable channels and a list of *Channels* whose description contains a certain keyword. In the client, build a binary search tree and test all those methods. Include code in the client to build a tree of at least 10 elements.

15. Write a program that includes a *BinarySearchTree* class that stores information about people. A *Person* is defined as having an email, an age, and a country of origin. Order the binary search tree based on the email of people. Add a method to search based on email. Add methods to search based on age or country of origin and return an *ArrayList* of *Person* object references. Add another method to return a list of *Persons* who are younger than a certain age. In the client, build a binary search tree and test all those methods based on user input. Include code in the client to build a tree of at least 10 elements.

15.10.8 Technical Writing

1. Consider the two search methods in exercise #60 that return a single *Channel* object reference; discuss their respective running times and the impact on deciding the ordering criteria on those running times.
2. A *heap* is a special type of complete binary tree. Research what a heap is, give a description of it, and give an example (i.e., draw it) of a heap with at least 10 elements. Research how to insert into a heap, and insert successively two elements (i.e., redraw the heap after each insertion) in your heap of 10 elements.

15.10.9 Group Project (for a group of 1,2, or 3 students)

1. A composite number is an integer that has exactly two factors, both of them prime. For example, 91(13 times 7) and 85 (17 times 5) are composite numbers. The RSA encryption algorithm is based on the fact that factoring is difficult; in particular, composite numbers are difficult to

factor (when they are large). Build a *BinarySearchTree* class that stores composite numbers and their two factors. The ordering of the tree should be based on the number (not the two factors). Include the appropriate methods to rebalance such a binary search tree so that it is height-balanced. Since you will store composite numbers, you should have a class encapsulating a composite number. Inside that class, provide a method to return a composite number object, including the two factors, based on the value of a number. In the client, build a very large binary search tree (you will need to use looping to build it). Note that you do not need to factor any composite number to fill the tree. For example, with 13 and 7, you can create the composite number 91 and store it along with its two factors, 13 and 7. Additionally, test the two ways to factor a composite number: one by using the binary search tree to retrieve its two factors, the other by factoring the composite number (you will need to create an algorithm for that). For a very large number, compare the actual running time of those two ways to factor.

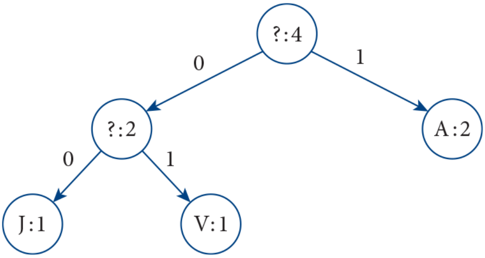
2. You need to encode a word input by the user into a binary *String* (made of 0s and 1s) that is minimal in length. You need to implement the following algorithm:

    - Create an *ArrayList* of objects for that *String:* each object has two attributes: a character (that is found in the *String)* and its frequency within the *String.* For example, if the user input is JAVA, there will be three objects in the list: J:1, A:2, and V:1 (Note: character:frequency is simply a *String* representation of such an object).
    - Order that *ArrayList* in ascending order by frequency. If the user input is JAVA, the ordered list of objects is J:1, V:1, A:2.
    - From that ordered *ArrayList,* create a parallel *ArrayList* of binary trees. Each binary tree contains only one object.

- Loop to combine the binary trees in that list two at a time until there is only one binary tree left in the *ArrayList.* At each iteration of the loop, the two binary trees with the lowest frequency are combined. When combining two binary trees, the frequency of the resulting binary tree is the sum of the frequencies of the two binary trees. If the user input is JAVA, there are two iterations. At the first iteration, the binary trees containing J:l and VI are combined, resulting in a binary tree with frequency 2 (its root can store an object containing a character of your choice, for example ?, and 2). At the second iteration, that binary tree is combined with the binary tree containing A:2, resulting in a binary tree with frequency 4 (again, its root can store an object containing a character of your choice, for example ?, and 4).
- The resulting/final binary tree contains the codes for each character in the original *String.* The characters are stored in the leaves. Each edge in the tree can be labeled 0 or 1; for example, an edge from a node to its left child can be labeled with a 0, and an edge from a node to its right child can be labeled with a 1. The *String of* 0s and Is that make up the path to a leaf is the encoding of the character stored in that leaf. If the user input is JAVA, a possible resulting binary tree is shown below; the encoding for J is 00, the encoding for V is 01, and the encoding for A is 1. Traverse the tree in order to find the encoding for each character in the *String.*
- Use the encoding scheme above to output the encoding for the word input by the user. Using the encoding above for J, V, and A, the encoded *String* for

JAVA is 001011. Note that the encoding is not unique (for example if you flip the 0s and the 1s, you have another solution).

- Advice: test your code at every step. Draw the tree on paper and check your result against your drawing. Be sure to test your code with many user inputs.



Description

© Lois GoBe/Shutterstock

Chapter 16

Graphical User Interfaces

CHAPTER CONTENTS

Introduction
16.1 GUI Applications Using *JFrame*
16.2 GUI Components
16.3 A Simple Component: *JLabel*
16.4 Event Handling
16.5 Text Fields
16.6 Command Buttons
16.7 Radio Buttons and Checkboxes
16.8 **Programming Activity 1: Working with Buttons**
16.9 Combo Boxes
16.10 Adapter Classes
16.11 Mouse Movements
16.12 Layout Managers: *GridLayout*
16.13 Layout Managers: *BorderLayout*
16.14 Using Panels to Nest Components
16.15 **Programming Activity 2: Working with Layout Managers**
16.16 Chapter Summary
16.17 Exercises, Problems, and Projects

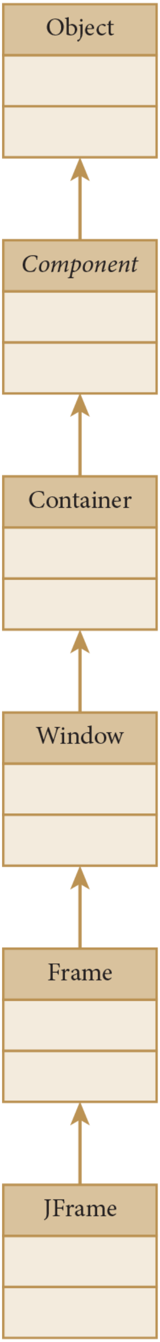
16.17.1 Multiple Choice Exercises
16.17.2 Reading and Understanding Code
16.17.3 Fill In the Code
16.17.4 Identifying Errors in Code
16.17.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
16.17.6 Write a Short Program
16.17.7 Programming Projects
16.17.8 Technical Writing
16.17.9 Group Project

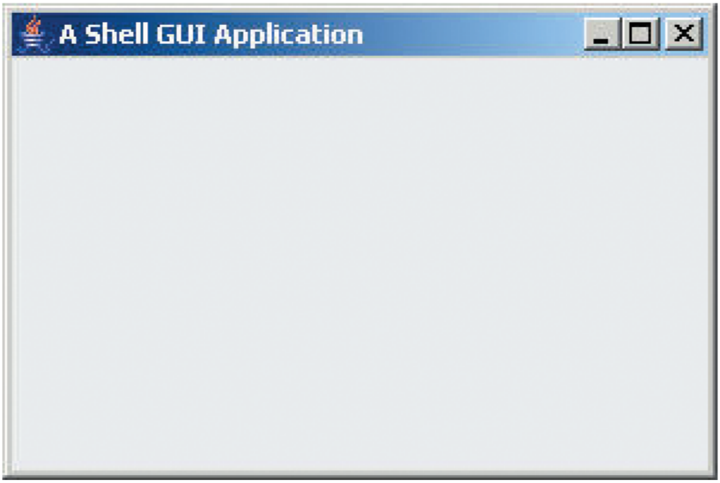
Introduction

Many applications we use every day have a graphical user interface, or GUI (pronounced goo-ey), which allows us to control the operation of the program visually. For example, web browsers, such as Microsoft Internet Explorer, provide menus, buttons, drop-down lists, and other visual input and output devices to allow the user to communicate with the program through mouse clicks, mouse movements, and by typing text into boxes.

GUIs enable the user to select the next function to be performed, to enter data, and to set program preferences, such as colors or fonts. GUIs also make a program easier to use because a GUI is a familiar interface to users. If you design your program's GUI well, users can quickly learn to operate your program, in many cases, without consulting documentation or requiring extensive training.

Java is very rich in GUI classes. In this chapter, we will present some of those classes, along with the main concepts associated with developing a GUI application.


Object
Component
Container
Window
Frame
JFrame



16.1 GUI Applications Using *JFrame*

The first step in developing a GUI application is to create a window. We will use the *JFrame* class in the *javax.swing* package to create a window for our GUI applications. The inheritance hierarchy for the *JFrame* class is shown in **Figure 16.1**.

Figure 16.1 Inheritance Hierarchy for *JFrame*

Description

Inheriting directly from *Object* is the *Component* class, which represents a graphical object that can be displayed. Inheriting from the *Component* class is the *Container* class, which, as its name indicates, can hold other objects. This is good because we will want to add GUI components to our window. The *Window* class, which inherits from the *Container* class, can be used to create a basic window. The *Frame* class, which inherits from the *Window* class, adds a title bar and a border to the window. The title bar contains icons that allow the user to minimize, maximize, resize, and close the window. This is the type of window we are accustomed to seeing.

We need to step down one more level of inheritance, however, to the *JFrame* class before we are ready to create our

GUI application. A *JFrame* object is a *swing* component. Thus, the *JFrame* class provides the functionality of *Frame*, as well as support for the *swing* architecture. We'll explain more about the *swing* components later in this chapter. For now, Figure 16.1 shows us that a *JFrame* object is a *Component*, a *Container*, a *Window*, and a *Frame* by inheritance, so we can use a *JFrame* object to display a window that will hold our GUI components, and thus present our GUI to the user. Our typical GUI application class will be a subclass of *JFrame*, inheriting its functionality as a window and a container. Thus, the class header for our GUI applications will follow this pattern:

```
public class ClassName extends JFrame
```

Description

Table 16.1 shows two constructors and some important methods of the *JFrame* class, some of which are inherited from its superclasses. Because our GUI applications extend the *JFrame* class, our applications inherit these methods also.

Useful *JFrame* Constructors and Methods

Constructors

Class	Constructor
`JFrame`	`JFrame( )`
constructs a *JFrame* object, initially invisible, with no text in the title bar	
`JFrame`	`JFrame( String titleBarText )`
constructs a *JFrame* object, initially invisible, with *titleBarText* displayed in the window's title bar	

Methods

Return value	Method name and argument list
`Container`	`getContentPane( )`
returns the content pane object for this window	
`void`	`setDefaultCloseOperation( int operation )`
sets the default operation when the user closes this window, that is,	

when the user clicks on the X icon in the top-right corner of the window	
`void`	`setSize( int width, int height )`
sizes the window to the specified *width* and *height* in pixels	
`void`	`setVisible( boolean mode )`
displays this window if *mode* is *true*; hides the window if *mode* is *false*	

Description

Example 16.1 shows a shell GUI application class. This class demonstrates the general format for building GUI applications, but has no components.

Example 16.1 A Shell GUI Application Using *JFrame*

```
 1 /*  A Shell GUI Application
 2     Anderson, Franceschi
 3 */
 4
 5 import javax.swing.JFrame;
 6 import java.awt.Container;
 7 // other import statements here as needed
 8
 9 public class ShellGUIApplication extends JFrame
10 {
11    private Container contents;
12    // declare other instance variables
13
14    // constructor
15    public ShellGUIApplication ( )
16    {
17      // call JFrame constructor with title bar text
18      super( "A Shell GUI Application" );
19
20      // get container for components
21      contents = getContentPane ( );
22
23      // set the layout manager

24
25      // instantiate GUI components and other instance variables
26
27      // add GUI components to the content pane
28
29      // set original size of window
30      setSize( 300, 200 );
31
32      // make window visible
33      setVisible( true );
34   }
35
36   public static void main( String [ ] args )
37   {
38     ShellGUIApplication basicGui = new ShellGUIApplication( );
39     basicGui.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
40   }
41 }
```

Description

We have defined one instance variable, *contents*, which is a *Container* object reference (line 11). Our subsequent GUI applications will define instance variables for the components to be placed in the window, as well as instance variables to hold the data of the application as primitive data types or objects.

The *main* method is coded at lines 36–40. We start by instantiating an object of this application class (line 38), which invokes the constructor (lines 14–34).

At line 39, we call the *setDefaultCloseOperation* method. The *EXIT_ON_CLOSE* argument is a useful *static* constant of the *JFrame* class, which specifies to the *setDefaultCloseOperation* method that the GUI application should terminate when the user closes the window.

A constructor in a GUI application has several tasks to perform:

- Call the constructor of the *JFrame* superclass.
- Get an object reference to the content pane container. We will add our GUI components to the content pane.
- Set the layout manager. Layout managers arrange the GUI components in the window.
- Instantiate each component.
- Add each component to the content pane.
- Set the size of the window.
- Display the window.

In our constructor, at line 18, we call the *JFrame* constructor, passing as an argument, a *String* representing the text that we want to be displayed in the title bar of the window. As in any subclass, this must be the first statement in the constructor.

At line 21, we call the *getContentPane* method, inherited from the *JFrame* class, and assign its return value to *contents*. The *getContentPane* method returns a *Container* object reference; subsequent applications will add our GUI components to this container.

Although this shell application does not have any GUI components, we have inserted comments to indicate where the component-related operations should be performed. Later in this chapter, we show you how to perform these actions.

COMMON ERROR TRAP

Be sure to call the *setSize* method to set the initial dimensions of the window and call the *setVisible* method to display the window and its contents. Omitting the call to the *setSize* method will create a default *JFrame* consisting of a title bar only, although it will be resizable by the user. If you omit the call to the *setVisible* method, the window will not open when the application begins.

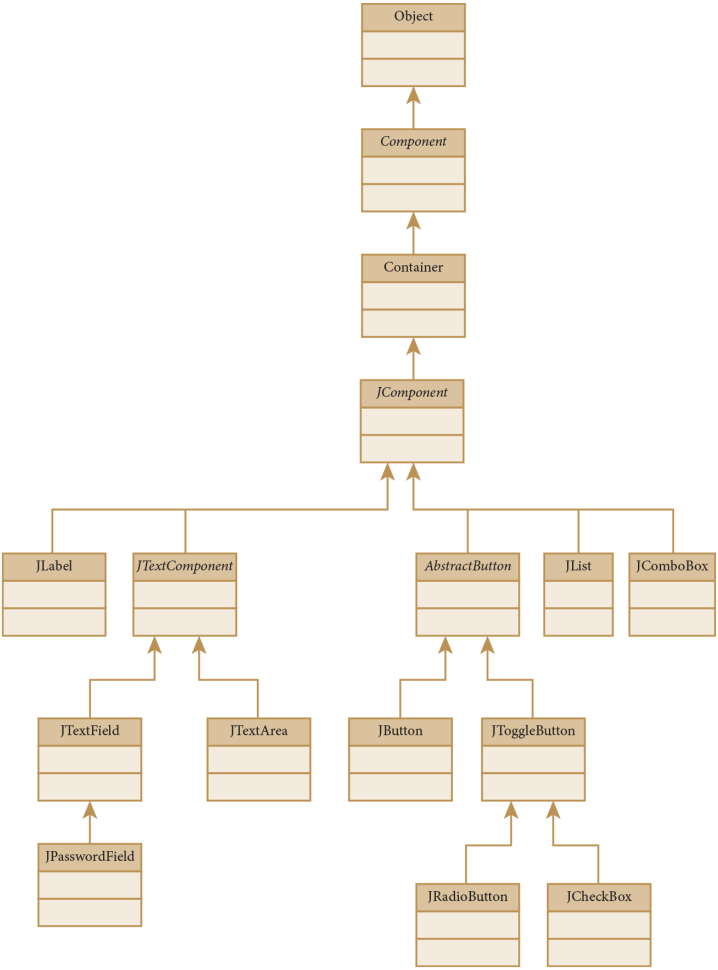
We then set the initial size of the window to a width of 300 pixels and a height of 200 pixels by calling the *setSize* method (line 30). Finally, we call the *setVisible* method (line 33) so that the window will be displayed when the constructor finishes executing. Calling the *setSize* and *setVisible* methods is not required, but if you omit either method call, you will get unfavorable results. If you omit the call to *setSize*, the window will consist of just the title bar. If you omit the call to *setVisible*, the window will not be displayed when the application begins.

The window generated by running Example 16.1 is shown in **Figure 16.2**. Note the text in the title bar, which was set by calling the *JFrame* constructor. When you run the program, try minimizing, maximizing, moving, resizing, and closing the window.

Figure 16.2 The Window Generated by Example 16.1

Description

We have opened a window successfully, but an empty window is not very impressive. In the next section, we will show you how to start adding GUI components to the window.


Object
Component
Container
JComponent
JLabel
JTextComponent
AbstractButton
JList
JComboBox
JTextField
JTextArea
JButton
JToggleButton
JPasswordField
JRadioButton
JCheckBox

16.2 GUI Components

We have already said that a **component** is an object having a graphical representation. Labels, text fields, buttons, radio buttons, checkboxes, and drop-down lists are examples of components. A component performs at least one of these functions:

- displays information
- collects data from the user

- allows the user to initiate program functions

Java provides an extensive set of classes that can be used to add a GUI to your applications. **Table 16.2** lists some GUI components and the Java classes that encapsulate them. All classes listed in Table 16.2 belong to the *javax.swing* package.

Selected GUI Components and Java Classes

Component	**Purpose**	**Java Class**
Label	Displays an image or read-only text. Labels are often paired with text fields and are used to identify the contents of the text field.	*JLabel*
Text field	A single-line text box for displaying information and accepting user input.	*JTextField*
Text area	Multiple-line text field for data entry or display.	*JTextArea*
Password field	Single-line text field for accepting passwords without displaying the characters typed.	*JPasswordField*
Button	Command button that the user clicks to signal that an operation should be performed.	*JButton*
Radio button	Toggle button that the user clicks to select one option in the group. Clicking on a radio button deselects any previously selected option.	*JRadioButton*
Checkbox	Toggle button that the user clicks to select 0, 1, or more options in a group.	*JCheckBox*

Drop-down list	Drop-down list of items that the user clicks to select one item.	*JComboBox*

Description

Java supports two implementations of GUI components: AWT (Abstract Window Toolkit) and *swing*.

The AWT components are the original implementation of Java components and hand off some of the display and behavior of the component to the native windowing system. In other words, on a PC, AWT components, such as buttons and frames, are rendered by the Windows windowing system; on a Macintosh computer, AWT components are rendered by the MacOS windowing system; on a Unix computer, AWT components are rendered by the X-Window system. As a result, the AWT GUI components automatically take on the appearance and behavior (commonly called the **look and feel**) of the windowing system on which the application is running. One disadvantage of AWT components, however, is that because of the inconsistencies in the look and feel among the various windowing systems, an application may behave slightly differently from one platform to another. Because AWT components rely on the native windowing system, they are often called **heavyweight** components.

The *swing* components are the second generation of GUI components, developed entirely in Java to provide a consistent look and feel from platform to platform. As such, they are often referred to as **lightweight** components. Some of the benefits of *swing* components are:

- Applications run consistently across platforms, which makes maintenance easier.
- The *swing* architecture has its roots in the model-view-controller paradigm, which facilitates programming:
    - the model represents the data for the application
    - the view is the visual representation of that data
    - the controller takes user input on the view and updates the model accordingly

- The *swing* components can be rendered in multiple windowing system styles, so your application can take on the look and feel of the platform on which it is running, if desired.

Given the advantages of *swing* over AWT, we will build our GUI applications using *swing* components exclusively. You can recognize a *swing* component because its class name begins with *J*. Thus, *JFrame* is a *swing* component. *Swing* components are found in the *javax.swing* package.

A summary of the inheritance hierarchy for selected *swing* components is shown in **Figure 16.3**. *JComponent* is the base class for all swing components, except for top-level containers such as *JFrame*. Notice that the *JFrame* class does not inherit from the *JComponent* class. However, the *JFrame* class and the *JComponents* shown in Table 16.2 do share some common superclasses: *Component* and *Container*. Thus, the *JComponents* are *Containers*, and as such, can contain other *JComponents* or other *Container* objects. This means that we can nest GUI components, which will help us organize the layout of our windows. We will show you how to do that later in the chapter.

Figure 16.3 The Inheritance Hierarchy for Some GUI Classes

Description

Table 16.3 lists some important methods of *JComponent*. Because these methods are *public*, the GUI components shown in Figure 16.3 inherit these methods.

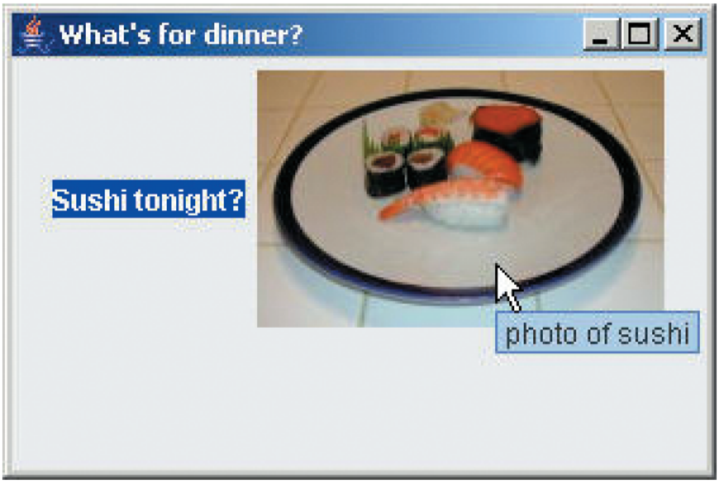
Some Useful *JComponent* Methods

Methods of the JComponent Class	
Return value	**Method name and argument list**
`void`	`setVisible (boolean mode )`

makes the component visible if *mode* is *true*; hides the component if *mode* is *false*. The default is visible.

`void`	`setToolTipText( String toolTip )`	sets the tool tip text to *toolTip*. When the mouse lingers over the component, the tool tip text will be displayed.
`void`	`setForeground ( Color foreColor )`	sets the foreground color of the component to *foreColor*.
`void`	`setBackground( Color backColor )`	sets the background color of the component to *backColor*.
`void`	`setOpaque ( boolean mode )`	sets the component's background to opaque if *mode* is *true*; sets the component's background to transparent if *mode* is *false*. If opaque, the component's background is filled with the component's background color; if transparent, the component's background is filled with the background color of the container on which it is placed. The default is transparent.
`void`	`setEnabled( boolean mode )`	enables the component if *mode* is *true*; disables the component if *mode* is *false*. An enabled component can respond to user input.

Description



16.3 A Simple Component: *JLabel*

Now it's time to place a component into the window. We will start with a simple component, a label, encapsulated by the *JLabel* class. A user does not interact with a label; the label just displays some information, such as a title, an identifier for another component, or an image.

Example 16.2 creates a window containing two labels, one that displays text and one that displays an image. **Figure 16.4** shows the window when the application is run.

Example 16.2 A Simple GUI Component: *JLabel*

```java
/*  Using JLabels to display text and images
    Anderson, Franceschi
*/

import java.awt.Container;
import javax.swing.JFrame;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import java.awt.FlowLayout;
import java.awt.Color;

public class Dinner extends JFrame
{
  private Container contents;
  private JLabel labelText;
  private JLabel labelImage;

  // Constructor
  public Dinner( )
  {
    super( "What's for dinner?" );       // call JFrame constructor

    contents = getContentPane( );        // get content pane

    contents.setLayout( new FlowLayout( ) ); // set layout manager

    // use the JLabel constructor with a String argument
    labelText = new JLabel( "Sushi tonight?" );

    // set label properties
    labelText.setForeground( Color.WHITE );
    labelText.setBackground( Color.BLUE );
    labelText.setOpaque( true );

    // use the JLabel constructor with an ImageIcon argument
    labelImage = new JLabel( new ImageIcon( "sushi.jpg" ) );

    // set tool tip text
    labelImage.setToolTipText( "photo of sushi" );

    // add the two labels to the content pane
    contents.add( labelText );
    contents.add( labelImage );

    setSize( 300, 200 );
    setVisible( true );
  }

  public static void main( String [ ] args )
  {
    Dinner dinner = new Dinner( );
    dinner.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
  }
}
```

Description

Figure 16.4 Running Example 16.2

Description

Java provides classes to help us organize window contents. These classes are called layout managers. They implement the *LayoutManager* interface and determine the size and position of the components within a container. We will use the simple *FlowLayout* layout manager for the next few examples; later in the chapter, we will look into two other, more sophisticated layout managers: *BorderLayout* and *GridLayout*.

The *FlowLayout* layout manager arranges components in rows from left to right in the order in which the components are added to the container. Whenever a newly added component does not fit into the current row, the *FlowLayout* layout manager starts a new row. With *FlowLayout*, if the user resizes the window, the components may be rearranged into fewer or more rows. We will use the default constructor of the *FlowLayout* class, which centers the components within each row.

To add components to a window, we use two methods of the *Container* class, shown in **Table 16.4**, to set the layout manager and add components to the window. Remember that we obtain the content pane of the *JFrame* window as a *Container* object. We call these methods using the content pane object reference.

Useful Methods of the *Container* Class

Methods of the Container Class

Return value	**Method name and argument list**
`void`	`setLayout( LayoutManager mgr )`
sets the layout manager of the window to *mgr*.	
`Component`	`add( Component component )`
adds the *component* to the container, using the rules of the layout manager. Returns the argument.	
`void`	`removeAll( )`
removes all components from the window.	

Description

In Example 16.2, we declare two *JLabel* instance variables at lines 15–16. At line 25, we set *FlowLayout* as the layout manager.

Next, we instantiate our *JLabel* objects. **Table 16.5** shows several constructors of the *JLabel* class.

Useful Constructors of the *JLabel* Class

Constructors of the JLabel Class

Class	Constructor
`JLabel`	`JLabel( String text )`
creates a *JLabel* object that displays the specified text.	
`JLabel`	`JLabel( String text, int alignment )`
creates a *JLabel* object that displays the specified text. The *alignment* argument specifies the alignment of the text within the label component. The *alignment* value can be any of the following *static int* constants from the *SwingConstants* interface: LEFT, CENTER, RIGHT, LEADING, or TRAILING. By default, the label text is left-adjusted.	
`JLabel`	`JLabel( Icon image )`
creates a *JLabel* object that displays the *image*.	

Description

The *SwingConstants* interface, which is implemented by many *JComponents* that display text, such as labels, text fields, and buttons, provides a collection of *static int* constants that can be used for positioning and orientation of the text within a component.

REFERENCE POINT

More information about the *SwingConstants* interface can be found on the Oracle Java website: *www.oracle.com/technetwork/java*.

The first *JLabel* object reference, *labelText*, is instantiated at line 28, using the constructor with a *String* argument.

The second *JLabel* object reference, *labelImage*, is instantiated at line 36, using the constructor that takes an *Icon* argument. The *ImageIcon* class encapsulates an image and implements the *Icon* interface, so an *IconImage* object can be used wherever an *Icon* argument is required. One of the *ImageIcon* constructors takes as a *String* argument the file name where the image is stored. Its API is:

```
public ImageIcon( String filename )
```

Description

Lines 31–33 call the *setForeground*, *setBackground*, and *setOpaque* methods inherited from the *JComponent* class to set some properties of *labelText*. On line 39, we set up a tool tip that will pop up the message “photo of sushi” when the user pauses the mouse pointer over the image. You can see this tool tip displayed in Figure 16.4.

COMMON ERROR TRAP

As for any object reference, it is mandatory to instantiate a component before using it. Forgetting to instantiate a component before adding it to the content pane will result in a *NullPointerException* at run time.

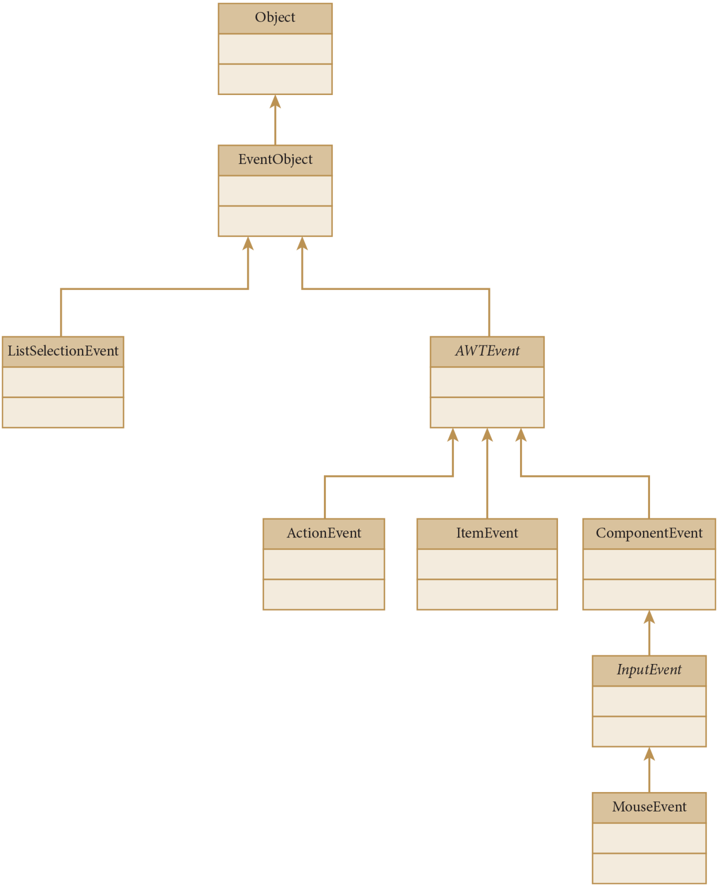
In lines 42–43, we add the two labels to the window. Because we are using the *FlowLayout* layout manager, the labels appear in the order we added them to the component, first *labelText*, then *labelImage*, centered left to right in the window. If the user resizes the window to be about the same width as the *JLabel* *labelText*, the layout manager will arrange the display so that the image appears under the text (as a second row).

Note that the last statement of the constructor (line 46) is a call to the *setVisible* method to make the window visible.

with these end-of-chapter questions

16.17.1	Multiple Choice Exercises Questions 1, 2, 3, 4

16.17.2	Reading and Understanding Code Question 18
16.17.3	Fill In the Code Question 27
16.17.4	Identifying Errors in Code Question 42
16.17.5	Debugging Area Questions 46, 47


Object
EventObject
ListSelectionEvent
AWTEvent
ActionEvent
ItemEvent
ComponentEvent
InputEvent
MouseEvent

16.4 Event Handling

Now we know how to open a window in an application, and we know how to display labels and images. But the user can't interact with our application yet, except to display a tool tip. We need to add some interactive GUI components, like buttons or text fields or lists. By interacting with these components the user will control which operations of the program will take place, and in what order.

GUI programming uses an **event-driven model** of programming, as opposed to the procedural model of programming we have been using thus far. By that, we mean that by using a GUI, we put the user in interactive control. For example, we might display some text fields, a few buttons, and a selectable list of items. Then our program will “sit back” and wait for the user to do something. When the user enters data into a text field, presses a button, or selects an item from the list, our program will respond, perform the function that the user has requested, then sit back again and wait for the user to do something else. These user actions generate **events**. Thus, the processing of our program will consist of responding to events caused by the user interacting with GUI components.

When the user interacts with a GUI component, the component **fires an event**. Java provides interfaces, classes, and methods to handle these events. Using these Java tools, we can register our application’s interest in being notified when an event occurs for a particular component, and we can specify the code we want to be executed when that event occurs.

To allow a user to interact with our application through a GUI component, we need to perform the following functions:

1. write an event handler class (called a **listener**)
2. instantiate an object of that class
3. register that listener on one or more components

A typical event handler class *implements* a listener interface. The listener interfaces, which inherit from the *EventListener* interface, are supplied in the *java.awt.event* and *javax.swing.event* packages. A listener interface specifies one or more abstract methods that an event handler class needs to implement. The listener methods receive as a parameter an event object, which represents the event that was fired.

An application can instantiate multiple event handlers, and a single event handler can be the listener for multiple components.

Event classes are subclasses of the *EventObject* class, as shown in **Figure 16.5**, and are in the *java.awt.event* and *javax.swing.event* packages. From the *EventObject* class, the event classes inherit the *getSource* method, which returns the object reference of the component that fired the event. Its API is shown in **Table 16.6**.

Figure 16.5 Event Class Hierarchy

Description

The *getSource* Method Inherited by Event Classes

Return Value	**Method Name and Argument List**
`Object`	`getSource( )`
returns the object reference of the component that fired the event	

Description

Thus, with some simple *if/else* statements, an event handler that is registered as the listener for more than one component can identify which of the components fired the event and decide on the appropriate action.

The types of listeners that we can register on a component depend on the types of events that the component fires. **Table 16.7** shows some user activities that generate events, the type of event object created, and the appropriate listener interface for the event handler to implement. Some of these components can fire other events, as well.

Component Events and Listeners

JComponent	**User Interaction That Generates an Event**	**The Event Object Created**	**Listener Interface the Event Handler Should Implement**
JTextField	Pressing Enter	*ActionEvent*	*ActionListener*
JButton	Pressing the button	*ActionEvent*	*ActionListener*
JRadioButton	Selecting a radio button	*ItemEvent*	*ItemListener*
JCheckBox	Selecting or deselecting a checkbox	*ItemEvent*	*ItemListener*
JList	Selecting an item	*ListSelectionEvent*	*ListSelectionListener*
JComboBox	Selecting an item	*ItemEvent*	*ItemListener*
Any component	Pressing or releasing mouse buttons	*MouseEvent*	*MouseListener*
Any component	Moving or dragging the mouse	*MouseEvent*	*MouseMotionListener*

Description

To add event handling to our *ShellGUIApplication* class, we will follow the following pattern. The code we need to add is shown in bold.

```
// import statements as needed

public class ClassName1 extends JFrame
{
  // declare components and other instance variables

  // constructor
  public ClassName1( )
  {
     // call JFrame constructor
     // get content pane
     // set the layout manager
     // instantiate components and other instance variables
     // add components to the content pane
     // declare and instantiate event handler objects
     // register event handlers on components

     // set window size
     // make window visible
  }

  // private event handler class
  private class EventHandlerName implements ListenerName
  {
     // implement the methods of the listener interface
     // to process the events
  }

  public static void main( String [ ] args )
  {
     // instantiate application object
  }
}
```

Description

Although we can define an event handler class as *public* in its own source file, we usually declare an event handler as a ***private* inner class** within the GUI application class. A *private* inner class is defined within the *public* class and has access to all the members of the *public* class. Thus, declaring our event handler as a *private* inner class simplifies our code by giving the event handler direct access to our application’s GUI components.

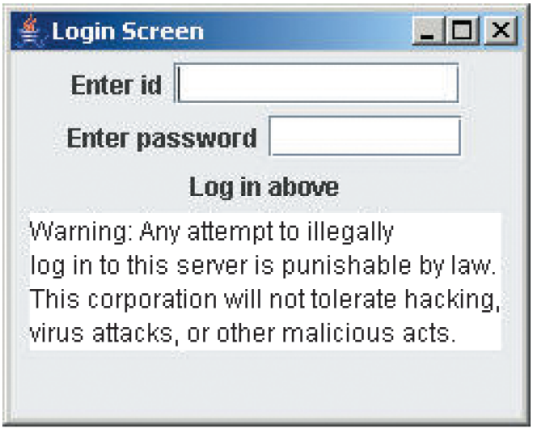
In the constructor, we instantiate an object of our event handler class. Then we register that event handler on a component by calling an *add…Listener* method. **Table 16.8** shows a few of these methods. We call the *add…Listener* method using the object reference of the component on which we want to register the listener and passing as an argument the event handler object we instantiated.

Some *add…Listener* Methods

***add…Listener* Method APIs**

```
void addActionListener( ActionListener
handler )
void addItemListener( ItemListener handler
)
void addListSelectionListener(
ListSelectionListener handler )
void addMouseListener( MouseListener
handler )
void addMouseMotionListener(
MouseMotionListener handler )
```

Description

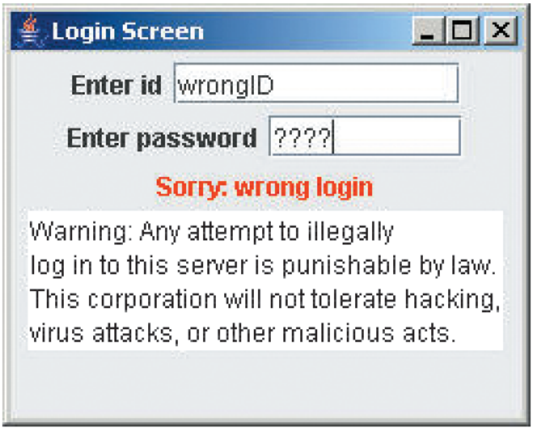


Enter id

Enter password

Log in above

Warning: Any attempt to illegally
log in to this server is punishable by law.
This corporation will not tolerate hacking,
virus attacks, or other malicious acts.

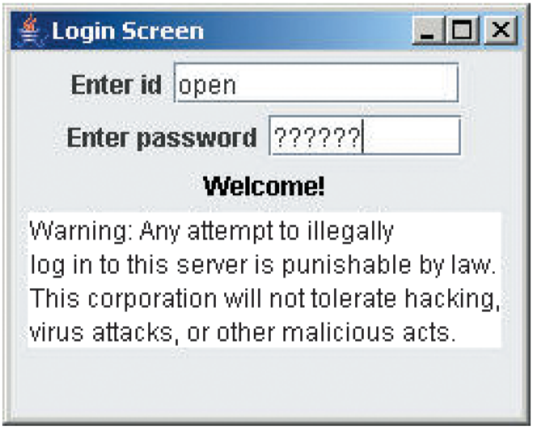


Enter id wrongID

Enter password ????

Sorry: wrong login

Warning: Any attempt to illegally
log in to this server is punishable by law.
This corporation will not tolerate hacking,
virus attacks, or other malicious acts.



Enter id open

Enter password ??????

Welcome!

Warning: Any attempt to illegally
log in to this server is punishable by law.
This corporation will not tolerate hacking,
virus attacks, or other malicious acts.

16.5 Text Fields

Now we are ready for our first GUI program with user interaction. For this first example, we will build a GUI that takes an ID and a password from a user. If the ID is “open” and the password is “sesame”, we display “Welcome!” Otherwise, we display “Sorry: wrong login”.

We will use a *JTextField* for the ID and a *JPasswordField* for the password. For confidentiality of passwords, the *JPasswordField* does not display the characters typed by the user. Instead, as each character is typed, the *JPasswordField* displays an echo character. The default echo character is *, but we can specify a different character, if desired. We will also use a *JTextArea* to display a legal warning to potential hackers. Both *JTextField* and *JTextArea* are direct subclasses of *JTextComponent*, which encapsulates text entry from the user. *JTextField* displays a single-line field, and *JTextArea* displays a multiline field.

Table 16.9 lists some useful constructors and methods of these text classes.

Useful Constructors and Methods of *JTextField*, *JTextArea*, and *JPasswordField*

Constructors and Methods of the *JTextField*, *JTextArea*, and *JPasswordField* Classes

Constructors

Class	Constructor
`JTextField`	`JTextField( String text, int numColumns )`
constructs a text field initially filled with *text*, with the specified number of columns.	
`JTextField`	`JTextField( int numberColumns )`
constructs an empty text field with the specified number of columns.	
`JTextArea`	`JTextArea( String`

`text )`

constructs a text area initially filled with *text.*

`JTextArea` `JTextArea( int numRows, int numColumns )`

constructs an empty text area with the number of rows and columns specified by *numRows* and *numColumns.*

`JTextArea` `JTextArea( String text, int numRows, int numColumns )`

constructs a text area initially filled with text, and with the number of rows and columns specified by *numRows* and *numColumns.*

`JPasswordField` `JPasswordField( int numberColumns )`

constructs an empty password field with the specified number of columns.

Methods Common to *JTextField*, *JTextArea*, and *JPasswordField* Classes

Return value **Method name and argument list**

`void` `setEditable( boolean mode )`

sets the properties of the text component as editable or noneditable, depending on whether *mode* is *true* or *false*. The default is editable.

`void` `setText( String newText )`

sets the text of the text component to *newText*.

`String` `getText( )`

returns the text contained in the text component.

Additional Methods of the *JPasswordField* Class

Return value **Method name and argument list**

`void` `setEchoChar( char c )`

sets the echo character of the password field to *c*.

`char [ ]` `getPassword( )`

returns the text entered in this password field as an array of *chars*. Note: This method is preferred over the *getText* method for getting the password typed by the user.

Description

Example 16.3 implements our login application, and **Figure 16.6** shows the application in action.

Example 16.3 The *Login* Class

```java
/* Using JTextFields, JTextArea, and JPasswordField
   Anderson, Franceschi
*/

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JTextArea;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.Color;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class Login extends JFrame
{
 private Container contents;
 private JLabel idLabel, passwordLabel, message;
 private JTextField id;
 private JPasswordField password;
 private JTextArea legal;

 // Constructor
 public Login( )
 {
  super( "Login Screen" );
  contents = getContentPane( );
  contents.setLayout( new FlowLayout( ) );

  idLabel = new JLabel( "Enter id" ); // label for ID
  id = new JTextField( "", 12 );      // instantiate ID text field

  passwordLabel = new JLabel( "Enter password" ); // password label
  password = new JPasswordField( 8 ); // instantiate password field
  password.setEchoChar( '?' );        // set echo character to '?'

  message = new JLabel( "Log in above" );  // label to hold messages

  // instantiate JTextArea with legal warning
  legal = new JTextArea( "Warning: Any attempt to illegally\n"
           + "log in to this server is punishable by law.\n"
           + "This corporation will not tolerate hacking,\n"
           + "virus attacks, or other malicious acts." );
  legal.setEditable( false );        // disable typing in this field

  // add all components to the window
  contents.add( idLabel );
  contents.add( id );
  contents.add( passwordLabel );
  contents.add( password );
  contents.add( message );
  contents.add( legal );

  // instantiate event handler for the text fields
  TextFieldHandler tfh = new TextFieldHandler( );

  // add event handler as listener for ID and password fields
  id.addActionListener( tfh );
  password.addActionListener( tfh );

  setSize( 250, 200 );
  setVisible( true );
 }

 // private inner class event handler
 private class TextFieldHandler implements ActionListener
 {
  public void actionPerformed( ActionEvent e )
  {
   if ( id.getText( ).equals( "open" )
    && ( new String( password.getPassword( ) ) ).equals( "sesame" ) )
   {
    message.setForeground( Color.BLACK );
    message.setText( "Welcome!" );
   }
   else
   {
    message.setForeground( Color.RED );
    message.setText( "Sorry: wrong login" );
   }
  }
 }

 public static void main( String [ ] args )
 {
  Login login = new Login( );
  login.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

Description

Figure 16.6 Running Example 16.3

Description

Lines 5–14 import the various classes that we use in this program. Line 16 declares the class, which we name *Login*, as a subclass of *JFrame*.

We declare our instance variables at lines 18–22. In addition to our three components (the *JTextField id*, the *JPasswordField password*, and the *JTextArea legal*), we also have three *JLabel* components (*idLabel, passwordLabel*, and *message*). The *idLabel* and *passwordLabel* components provide descriptions for the corresponding text fields. We will use the *message* label to display feedback to the user on whether the login was successful.

At line 32, we instantiate the *JTextField id* using the constructor that takes an initial value and the size of the text field. The initial value is blank, and we want the text field to accommodate 12 characters. At line 35, we instantiate the *JPasswordField password* and set its length to 8 characters. At line 36, we call the *setEchoChar* method to reset the echo character from its default of * to ?. Thus, when the user types characters into the password field, a ? will be displayed for each character typed.

At lines 40–44, we instantiate the *JTextArea legal*. By default, a user can enter data into a text area. However, since we are using this text area only to display a legal warning, we turn off the ability of the user to type into the text area by calling the *setEditable* method with the argument *false* (line 45).

At lines 47–53, we add all the components to the content pane. As Figure 16.6 shows, the components will be displayed in the window in the order we added them to the container.

After the window is displayed, our application will wait for the user to interact with the components. When the user enters a value into either the *id* or the *password* field, we want to check whether the values the user entered are correct. This will be the job of the event handler for those fields.

Pressing the *Enter key* in a text field generates an *ActionEvent*; therefore, at lines 66–83, we coded our event handler as the *private* inner class, *TextFieldHandler*, which *implements* the *ActionListener* interface. As shown in **Table 16.10**, the *ActionListener* interface has only one *abstract* method, *actionPerformed*, which receives an *ActionEvent* parameter.

ActionListener Method API

`public void actionPerformed( ActionEvent event )`

An event handler that implements the *ActionListener* interface writes code in this method to respond to the *ActionEvent* fired by any registered components.

Description

Be sure that you code the *actionPerformed* method correctly, as shown in Table 16.10. Failure to code the header properly causes a compiler error. For example, misspelling the method name generates the following compiler error:

```
Login.TextFieldHandler is not abstract and does not override
abstract method actionPerformed(ActionEvent) in ActionListener
private class TextFieldHandler implements ActionListener
        ^
1 error
```

Description

COMMON ERROR TRAP

Be sure that the header of the *actionPerformed* method is coded correctly. Otherwise, your method will not override the *abstract actionPerformed* method, as required by the *ActionListener* interface.

In the constructor, we declare and instantiate a *TextFieldHandler* object reference, *tfh* (line 55–56). Then at lines 58–60, we register *tfh* as the listener on the *id* and *password* components. When either *id* or *password* is in focus (that is, if the mouse was last clicked on the component) and the user presses the *Enter* key, the component that is in focus will fire an action event and our *actionPerformed* method will be executed.

We implement the *actionPerformed* method at lines 69–82. Because we defined this event handler as a *private* inner class, the *actionPerformed* method can directly access the *id* and *password* fields. At lines 71–72, we test if the *id* and *password* entered by the user are correct.

As shown in Table 16.9, the *getPassword* method of the *JPasswordField* class returns an array of *chars*. At line 72, to convert that array of *chars* to a *String*, we use a constructor of the *String* class with the following API, which takes an array of *chars* as an argument:

```
public String ( char [ ] charArray )
```

Description

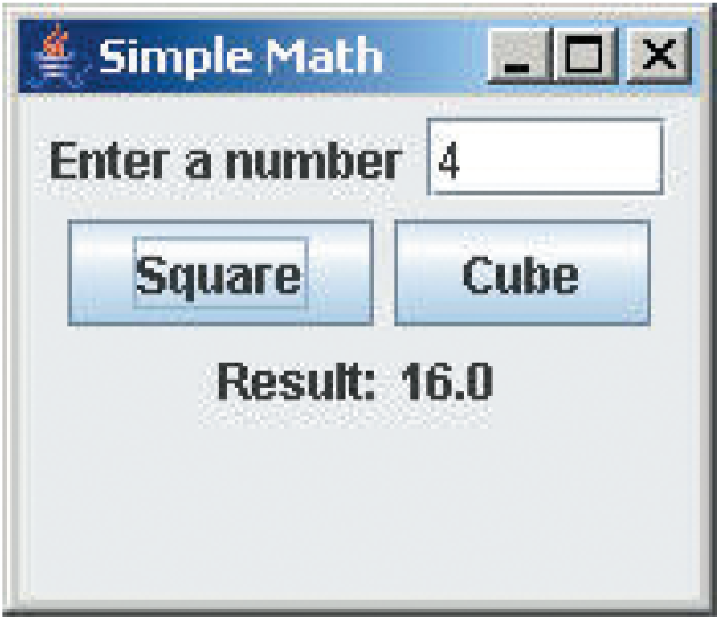
If the user has typed correct values into both the *id* and *password* fields, our event handler sets the foreground color to black and sets the text of *message* to “Welcome!” (lines 74–75). If either the id or the password is incorrect, we set the foreground color to red and set the text of *message* to “Sorry: wrong login” (lines 79–80).

COMMON ERROR TRAP

If you do not register a listener on a component, the component will not fire an event. Thus, the event handler will not execute when the user interacts with the component.

Note that we did not register our event handler for the *JTextArea*. We have disabled typing into that field, so the user cannot enter new text into that field. Also, if we did not register a listener on the *id* and *password* components, the user could still type characters into these two fields and press the *Enter* key, but no events would fire, and therefore, our application would not respond to the user’s actions.

Figure 16.6 shows the application when the window first opens, after the user enters incorrect login values, and after the user enters correct login values. We suggest you become familiar with this application. First run the application as it is written. Then, delete either line 59 or 60, which registers the listeners on the components, and test the application again. Finally, change the echo character and test the application again.



16.6 Command Buttons

Handling an event associated with a button follows the same pattern as Example 16.3. We instantiate an event handler and add that event handler as a listener for the button. Clicking on a button generates an *ActionEvent*, so our listener needs to implement the *ActionListener* interface.

In **Example 16.4**, we present the user with a text field and two command buttons. The user enters a number into the text field, then presses one of the two buttons. When the user clicks on the first button, we square the number; when the user clicks on the second button, we cube the number. We then display the result using a label component. To determine whether to square or cube the number, our event handler will need to identify which button was clicked by calling the *getSource* method of the *ActionEvent* class, as described in Table 16.6.

This example will demonstrate the following:

- how to handle an event originating from a button

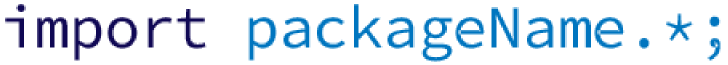
- how to determine which component fired the event

Example 16.4 Simple Math Using *JButton*

```java
/* Simple Math Operations Using JButtons
   Anderson, Franceschi
 */

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleMath extends JFrame
{
 private Container contents;
 private JLabel operandLabel, resultLabel, result;
 private JTextField operand;
 private JButton square, cube;

 public SimpleMath( )
 {
  super( "Simple Math" );
  contents = getContentPane( );
  contents.setLayout( new FlowLayout( ) );

  operandLabel = new JLabel( "Enter a number" ); // text field label
  operand = new JTextField( 5 ); // text field is 5 characters wide

  // instantiate buttons
  square = new JButton( "Square" );
  cube = new JButton( "Cube" );

  resultLabel = new JLabel( "Result:" ); // label for result
  result = new JLabel( "???" );          // label to hold result

  // add components to the window
  contents.add( operandLabel );
  contents.add( operand );
  contents.add( square );
  contents.add( cube );
  contents.add( resultLabel );
  contents.add( result );

  // instantiate our event handler
  ButtonHandler bh = new ButtonHandler( );

  // add event handler as listener for both buttons
  square.addActionListener( bh );
  cube.addActionListener( bh );

  setSize( 175, 150 );
  setVisible( true );
 }

 // private inner class event handler
 private class ButtonHandler implements ActionListener
 {
  // implement actionPerformed method
  public void actionPerformed( ActionEvent ae )
  {
   try
   {
    double op = Double.parseDouble( operand.getText( ) );

    // identify which button was pressed
    if ( ae.getSource( ) == square )
      result.setText( Double.valueOf( op * op ).toString( ) );
    else if ( ae.getSource( ) == cube )
      result.setText( Double.valueOf( op * op * op ).toString( ) );
   }
   catch ( NumberFormatException e )
   {
    operandLabel.setText( "Enter a number" );
    operand.setText( "" );
    result.setText( "???" );
   }
  }
 }

 public static void main( String [ ] args )
 {
  SimpleMath sm = new SimpleMath( );
  sm.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

Description

As you have seen in the previous examples, GUI applications use a number of Java classes. For Example 16.4, as well as the remaining examples in this chapter, we will use the bulk *import* statement, which replaces the individual class names with the wildcard character (*). Its syntax is:



```
import packageName.*;
```

Description

In this way, we write only one *import* statement per package (lines 5–7).

Notice that the *java.awt.event* package is different from the *java.awt* package. You might assume that importing *java.awt*.* also imports the classes in the *java.awt.event* package. That is not the case, however; we need to import both packages.

COMMON ERROR TRAP

The *java.awt.event* package is not imported with the *java.awt package*. Include import statements for both the *java.awt* and the *java.awt.event* packages.

At lines 51–74, we define our event handler, *ButtonHandler*, as a *private* inner class. Because clicking a button fires an *ActionEvent*, our event handler implements the *ActionListener* interface. Thus, we provide our code to handle the event in the *actionPerformed* method. Line 59 retrieves the data the user typed into the text field and attempts to convert the text to a *double*. Because the *parseDouble* method will throw a *NumberFormatException* if the user enters a value that cannot be converted to a *double*, we perform the conversion inside a *try* block.

The *if* statement at line 62 calls the *getSource* method using the *ActionEvent* object reference to test if the component that fired the event is the *square* button. If so, we set the text of the *result* label to the square of the number entered.

If the *square* button did not fire the event, we test if the source of the event was the *cube* button (line 64). If so, we set the text of the *result* label to the cube of the number entered. Actually, the test in line 64 is not necessary, because we registered our event handler as the listener for only two buttons.

So if the source isn't the *square* button, then the *cube* button must have fired the event. However, in order to improve readability and maintenance, it is good practice to check the source of the event specifically, in particular if additional buttons or components will be added to the application later.

SOFTWARE ENGINEERING TIP

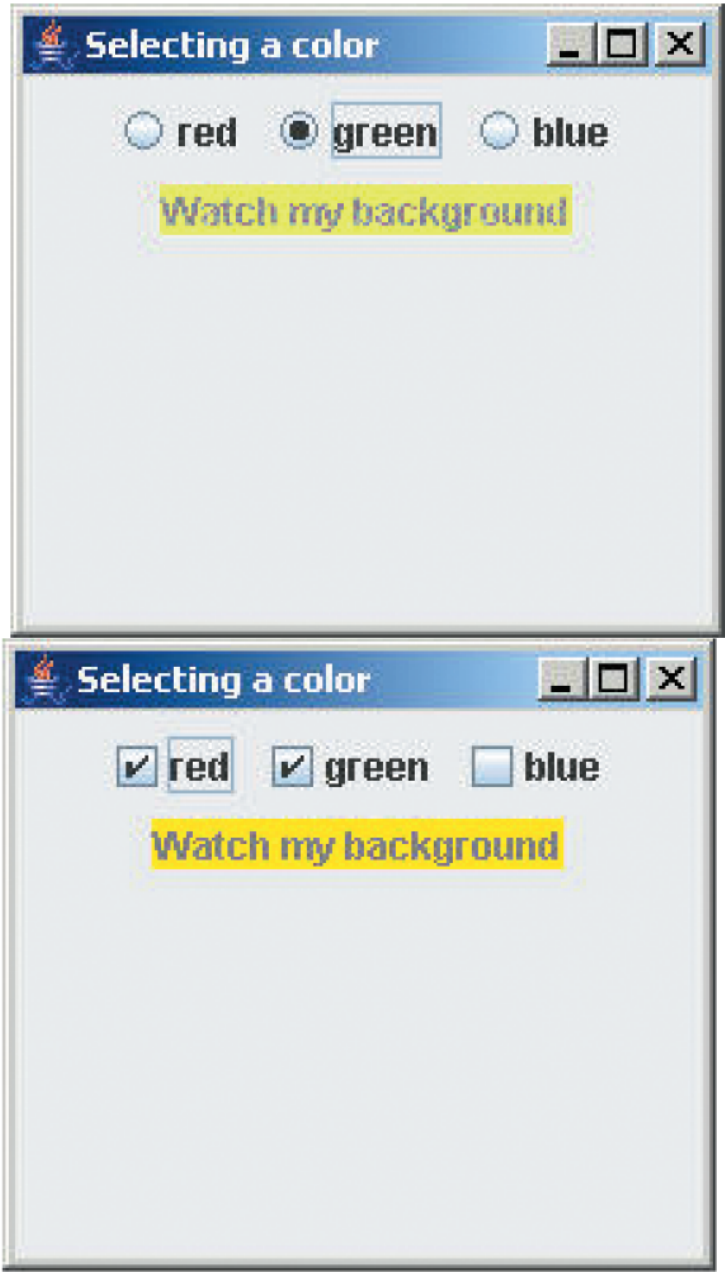
Verify the source component before processing the event.

In the constructor, at lines 40–41, we declare and instantiate the *ButtonHandler* event handler, *bh*. At lines 43–45, we call the *addActionListener* method to register the handler on our two buttons.

The output of this example is shown in **Figure 16.7**. When you run this example, try entering a valid number into the text field, and press each button. Then try to enter text that cannot be converted to a number. Also, try deleting either statement that registers the event listener on the buttons (line 44 or 45), and run the application again.

Figure 16.7 Running Example 16.4

Description


Selecting a color
○ red ● green ○ blue
Watch my background
Selecting a color
☑ red ☑ green ☐ blue
Watch my background

16.7 Radio Buttons and Checkboxes

If you have ever completed a survey on the web, you are probably acquainted with radio buttons and checkboxes.

Radio buttons prompt the user to select one of several mutually exclusive options. Clicking on any radio button deselects any previously selected option. Thus, in a group of radio buttons, a user can select only one option at a time.

Checkboxes are often associated with the instruction “check all that apply”; that is, the user is asked to select 0, 1, or more options. A checkbox is a toggle button in that if the option is not currently selected, clicking on a checkbox selects the option, and if the option is currently selected, clicking on the checkbox deselects the option.

The *JRadioButton* and *JCheckBox* classes, which belong to the *javax.swing* package, encapsulate the concepts of radio buttons and checkboxes, respectively. We will present two similar examples in order to illustrate how to use these classes and how they differ. Both examples allow the user to select the background color for a label component. We display three color options: red, green, and blue. Using radio buttons, only one option can be selected at a time. Thus, by clicking on a radio button, the user will cause the background of the label to be displayed in one of three colors. Using checkboxes, the user can select any combination of the three color options, so the label color can be set to any of eight possible combinations. **Table 16.11** lists all these combinations.

Selecting Color Using Radio Buttons vs.Checkboxes

Using Radio Buttons
(1 selection possible at a time)

Color Selection			**Resulting Color**
red	**green**	**blue**	
●	○	○	red
○	●	○	green
○	○	●	blue

Using Checkboxes
(0 to 3 selections possible at a time)

Color Selections			**Resulting Color**
red	**green**	**blue**	
☐	☐	☐	black
☑	☐	☐	red
☐	☑	☐	green
☐	☐	☑	blue
☑	☑	☐	yellow
☑	☐	☑	purple
☐	☑	☑	blue-green
☑	☑	☑	white

Description

Table 16.12 shows several constructors of the *JRadioButton*, *ButtonGroup*, and *JCheckBox* classes.

Useful Constructors of the *JRadioButton*, *ButtonGroup*, and *JCheckBox* Classes

Constructors of the *JRadioButton*, *ButtonGroup*, and *JCheckBox* Classes

Constructors

Class	Constructor
`JRadioButton`	`JRadioButton( String buttonLabel )` constructs a radio button labeled *buttonLabel*. By default, the radio button is initially deselected.
`JRadioButton`	`JRadioButton( String buttonLabel, boolean selected )` constructs a radio button labeled *buttonLabel*. If *selected* is *true*, the button is initially selected; if *selected* is *false*, the button is deselected.
`ButtonGroup`	`ButtonGroup( )` constructs a checkbox labeled *checkBoxLabel*. By default, the checkbox is initially deselected.
`JCheckBox`	`JCheckBox( String checkBoxLabel, boolean selected )` constructs a checkbox labeled *checkBoxLabel*. If *selected* is *true*, the checkbox is initially selected; if *selected* is *false*, the checkbox is initially deselected.

Description

Selecting or deselecting radio buttons and checkboxes fires an *ItemEvent*. To receive this event, our event handler needs to implement the *ItemListener* interface, which has only one method, shown in **Table 16.13**.

***ItemListener* Method API**

`public void itemStateChanged( ItemEvent event )` An event handler that implements the *ItemListener* interface writes

code in this method to respond to the *ItemEvent* fired by any registered components.

Description

Note that selecting a radio button fires both an *ItemEvent* and an *ActionEvent*, so an alternative for radio buttons is to register a handler that implements the *ActionListener* interface. We have chosen to use an *ItemListener* for both radio buttons and checkboxes to demonstrate the similarities and differences between the two components.

Example 16.5 shows the color selection application using radio buttons.

Example 16.5 Selecting Colors Using *JRadioButtons*

```
/* Select a Color using JRadioButtons
   Anderson, Franceschi
*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ChangingColors extends JFrame
{
 private Container contents;
 private JRadioButton red, green, blue;
 private ButtonGroup colorGroup;
 private JLabel label;
 private Color selectedColor = Color.RED;

 public ChangingColors( )
 {
  super( "Selecting a color" );
  contents = getContentPane( );
  contents.setLayout( new FlowLayout( ) );

  red = new JRadioButton( "red", true );
  green = new JRadioButton( "green" );
  blue = new JRadioButton( "blue" );

  label = new JLabel( "Watch my background" );
  label.setForeground( Color.GRAY );
  label.setOpaque( true );
  label.setBackground( selectedColor );

  contents.add( red );
  contents.add( green );
  contents.add( blue );
  contents.add( label );

  // create button group
  colorGroup = new ButtonGroup( );
  colorGroup.add( red );
  colorGroup.add( green );
  colorGroup.add( blue );

  // create RadioButtonHandler event handler
  // and register it on the radio buttons
  RadioButtonHandler rbh = new RadioButtonHandler( );
  red.addItemListener( rbh );
  green.addItemListener( rbh );
  blue.addItemListener( rbh );

  setSize( 225, 200 );
  setVisible( true );
 }

 private class RadioButtonHandler implements ItemListener
 {
  public void itemStateChanged( ItemEvent ie )
  {
   if ( ie.getSource( ) == red )
      selectedColor = Color.RED;
   else if ( ie.getSource( ) == green )
      selectedColor = Color.GREEN;
   else if ( ie.getSource( ) == blue )
      selectedColor = Color.BLUE;

   label.setBackground( selectedColor );
  }
 }

 public static void main( String [ ] args )
 {
  ChangingColors cc = new ChangingColors( );
  cc.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

Description

The various instance variables are declared at lines 11–15. The instance variable *selectedColor*, which is initialized to red, will store the choice of the user and will be used to set the background of the *JLabel* component *label*. The three radio buttons and *label* are instantiated at lines 23–27. At line 23, we use a *JRadioButton* constructor with two arguments to instantiate the radio button *red*. The first argument is the title of the radio button; the second argument, *true*, specifies that *red* is selected. Thus, when the window appears, the *red* radio button will be selected. In order to reflect the effect of the selected radio button, we set the background color of the label to red at line 30.

At lines 24 and 25, *green* and *blue* are instantiated using the *JRadioButton* constructor with only a *String* argument, specifying the radio button titles. The four components are added to the content pane at lines 32–35.

At lines 37–41, we instantiate a *ButtonGroup* object, then add the *red*, *green*, and *blue* radio buttons to *colorGroup* so that they will be mutually exclusive selections.

At lines 43–48, we instantiate our event handler, *RadioButtonHandler rbh*, and call the *addItemListener* method to register the listener *rbh* on the three radio buttons.

At lines 54–67, the *private* class *RadioButtonHandler implements* the *ItemListener* interface. The only method of *ItemListener*, *itemStateChanged*, is overriden at lines 56–66. Inside *itemStateChanged*, we use an *if/else* statement to determine which radio button was selected in order to set *selectedColor* to the appropriate color.

COMMON ERROR TRAP

Forgetting to make *JRadioButtons* part of the same *ButtonGroup* makes them independent and not mutually exclusive.

Figure 16.8 shows a run of the example after the user has clicked on the *green* radio button. Try running the example and clicking on each radio button. Try modifying the code; for example, delete the lines adding the radio buttons to the button group, then compile and run the program again.

Figure 16.8 Running Example 16.5

Description

Example 16.6 shows the color selection application using checkboxes.

Example 16.6 Using *JCheckBoxes*

```java
/*  Using JCheckBoxes
    Anderson, Franceschi
*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MixingColors extends JFrame
{
 private Container contents;
 private JCheckBox red, green, blue;
 private int redValue, greenValue, blueValue;
 private JLabel label;

 public MixingColors( )
 {
  super( "Selecting a color" );
  contents = getContentPane( );
  contents.setLayout( new FlowLayout( ) );

  red = new JCheckBox( "red" );
  green = new JCheckBox( "green" );
  blue = new JCheckBox( "blue" );

  label = new JLabel( "Watch my background" );
  label.setOpaque( true );
  label.setForeground( Color.GRAY );
  label.setBackground( new Color ( 0, 0, 0 ) );

  contents.add( red );
  contents.add( green );
  contents.add( blue );
  contents.add( label );

  // create CheckBoxHandler event handler
  // and register it on the checkboxes
  CheckBoxHandler cbh = new CheckBoxHandler( );
  red.addItemListener( cbh );
  green.addItemListener( cbh );
  blue.addItemListener( cbh );

  setSize( 225, 200 );
  setVisible( true );
 }

 private class CheckBoxHandler implements ItemListener
 {
  public void itemStateChanged( ItemEvent ie )
  {
   if ( ie.getSource( ) == red )
   {
    if ( ie.getStateChange( ) == ItemEvent.SELECTED )
      redValue = 255;
    else
      redValue = 0;
   }
   else if ( ie.getSource( ) == green )
   {
    if ( ie.getStateChange( ) == ItemEvent.SELECTED )
      greenValue = 255;
    else
      greenValue = 0;
   }
   else if ( ie.getSource( ) == blue )
   {
    if ( ie.getStateChange( ) == ItemEvent.SELECTED )
      blueValue = 255;
    else
      blueValue = 0;
   }

   label.setBackground(
        new Color( redValue, greenValue, blueValue ) );
  }
 }

 public static void main( String [ ] args )
 {
  MixingColors mc = new MixingColors( );
  mc.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

Description

The various instance variables are declared at lines 11–14. The instance variables *redValue*, *greenValue*, and *blueValue* will store the red, green, and blue intensity values, depending on which checkboxes the user selects. These values will be used to set the background color of the *JLabel* component *label*.

The three checkboxes and *label* are instantiated at lines 22–26. We use a *JCheckBox* constructor with a *String* argument that specifies the checkbox titles. The four components are added to the content pane at lines 31–34.

At lines 36–41, we instantiate the *CheckBoxHandler* event handler and register the *cbh* listener on the three checkboxes by calling the *addItemListener* method for each checkbox.

At lines 47–76, we define the *private* inner class, *CheckBoxHandler*, which *implements* the *ItemListener* interface. The only method of *ItemListener*, *itemStateChanged*, is overriden at lines 49–75. Inside *itemStateChanged*, we first use an *if/else* statement to determine which checkbox fired the event. We then use a nested *if/else* statement to determine if that checkbox has just been selected. The *getStateChange* method in the *ItemEvent* class returns this information. Its API is shown in **Table 16.14**. The *ItemEvent* class provides the *static int* constants *SELECTED* and *DESELECTED* for convenience in coding this method call. Every click on our checkboxes will fire an event, so calling the *getStateChange* method helps us to distinguish between the event fired when the user selects a checkbox from the event fired when the user deselects a checkbox.

A Useful Method of the *ItemEvent* Class

Return Value	**Method Name and Argument List**
`int`	`getStateChange( )` returns the state of the checkbox. If the checkbox is selected, the value *SELECTED* is returned; if the checkbox is deselected, the value

	DESELECTED is returned, where *SELECTED* and *DESELECTED* are *static int* constants of the *ItemEvent* class.

Description

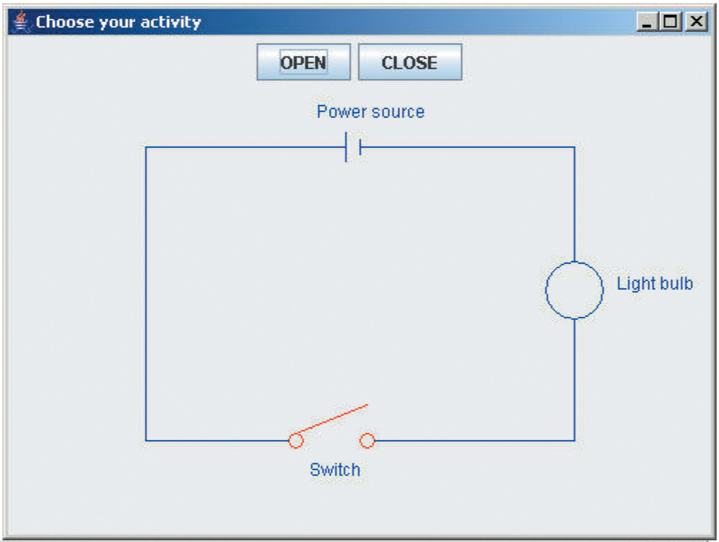
If the source checkbox was selected, we set the corresponding color value to 255 (lines 54, 61, and 68). If the source checkbox was deselected, we set the corresponding color value to 0 (lines 56, 63, and 70). For example, if the *red* checkbox is the source of the event, then we check whether the *red* checkbox was selected. If so, we set *redValue* to 255. Otherwise, the red checkbox must have been deselected, so we set *redValue* to 0.

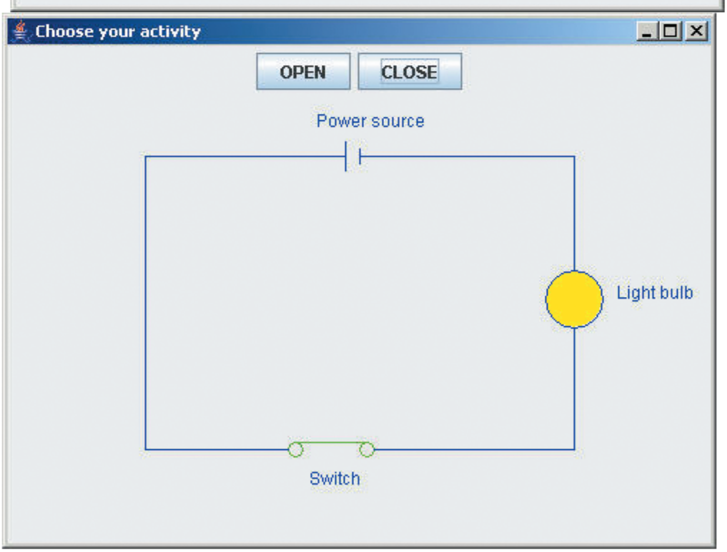
At lines 73–74, we set the background color of *label* to a color we instantiate using the values of *redValue*, *greenValue*, and *blueValue* for the color's respective intensities of red, green, and blue.

Figure 16.9 shows a run of the example after the user has selected the red and green checkboxes.

Figure 16.9 Running Example 16.6

Description


Choose your activity
OPEN
CLOSE
Power source
Light bulb
Switch


Choose your activity
OPEN
CLOSE
Power source
Light bulb
Switch

16.8 Programming Activity 1: Working with Buttons

In this activity, you will work with two *JButtons* that control one electrical switch. Specifically, you will write the code to perform the following operations:

1. If the user clicks on the “OPEN” button, open the switch.
2. If the user clicks on the “CLOSE” button, close the switch.

The framework for this Programming Activity will animate your code so that you can check its accuracy. **Figure 16.10** and **Figure 16.11** show the application after the user has clicked on the button labeled “OPEN” and the button labeled “CLOSE”, respectively.

Figure 16.10 User Clicked on “OPEN”

Description

Figure 16.11 User Clicked on “CLOSE”

Description

Instructions

Copy the source files in the Programming Activity 1 folder for this chapter to a folder on your computer. Note that all files should be in the same folder.

Open the *JButtonPractice.java* source file. Searching for five asterisks (*****) in the source code will position you to the first code section and then to the second location where you will add your code. In task 1, you will fill in the code to instantiate a button handler and register the handler on the appropriate components. In task 2, you will code the *private* class implementing the listener interface that handles the event generated when the user clicks on either button. Be careful: The class name you use for the handler in the first task needs to be the same as the name you use in the second task. It could be helpful to complete task 2 first. **Example 16.7** shows the section of the *JButtonPractice* source code where you will add your code.

Example 16.7 Location of Student Code In *JButtonPractice.java*

```
        ...
        open = new JButton( "OPEN" );
        contents.add( open );
        close = new JButton( "CLOSE" );
        contents.add( close );

        // ***** 1. Student code starts here
        // declare and instantiate the button handler
        // and register it on the buttons
        // end of task 1

        setSize( 500, 375 );
        setVisible( true );
    }

    // ***** 2. Student code restarts here
    // Code a private class to implement the correct Listener
    // and its required method
    // To open the switch, call the open method with the statement
    //   open( );
    // To close the switch, call the close method with the statement
    //   close( );
    // The last statement of the method should be
    //   animate( );


    // end of task 2
```

Description

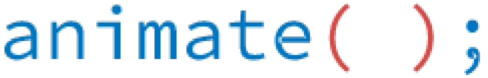
The framework will animate your code so that you can watch your code work. For this to happen, be sure that you call the *animate* method.

To test your code, compile and run the *JButtonPractice.java* file, then click on each button.

Troubleshooting

If your method implementation does not animate, check these tips:

- Verify that the last statement in your method inside the *private* class is:



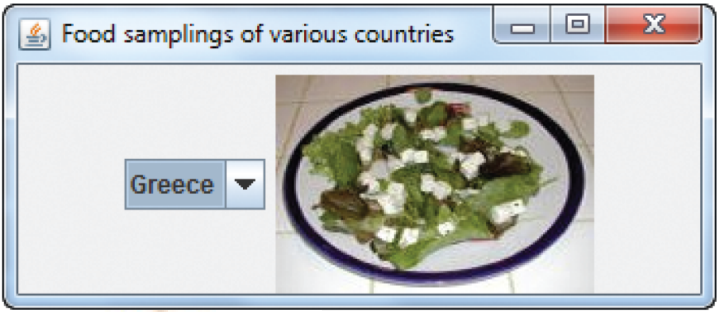
```
animate( );
```

- Verify that your listener is registered on the components.

- Verify that you have identified the origin of the event correctly.

DISCUSSION QUESTIONS

1. Explain why the *getSource* method is useful here.
2. Could you implement the preceding with *JRadioButtons* instead of *JButtons*? What interface would you implement? What method of the interface would you need to override? What type of event would you handle?



16.9 Combo Boxes

A *JComboBox* implements a drop-down list. When the combo box appears, one item is displayed, along with a button showing a down arrow. When the user presses on the button, the combo box “drops” open and displays a list of items, with a scroll bar for viewing more items. The user can select only one item from the list.

When the user selects an item, the list closes and the selected item is the one item displayed. A *JComboBox* fires an *ItemEvent*, so the event handler must implement the *ItemListener* interface, and thus, provide the *itemStateChanged* method.

Table 16.15 shows the APIs for a constructor and some useful methods of the *JComboBox* class. Note *JComboBox* is a generic class; thus, the constructor requires that you specify the class type of the items in the array that will make up the drop-down list. Often the class type is *String*. The *JComboBox* class provides the *getSelectedIndex* method, so that the event handler can determine which item the user has selected, and the *setSelectedIndex* method, which typically is used to initialize the default item displayed when the list appears. The *setMaximumRowCount* method allows us to specify how many items to display when the combo box opens. The user can use the scroll bar to move through all the items in the list.

Useful Constructors and Methods of the *JComboBox* Class

Constructors and Methods of the *JComboBox* Class

Constructor

Class	**Constructor**
`JComboBox`	`JComboBox( Object [ ] arrayName )`
constructs a new *JComboBox* component initially filled with the objects in *arrayName.* Often, the objects are *Strings*.	

Methods

Return value	**Method name and argument list**
`void`	`setSelectedIndex( int index )`
sets the item at *index* as selected. The index of the first item in the list is 0.	
`int`	`getSelectedIndex( )`
returns the index of the selected item. The index of the first item in the list is 0.	
`void`	`setMaximumRowCount( int size )`
sets the number of rows that will be visible at one time to *size*.	

Description

To illustrate how to use the *JComboBox* class, **Example 16.8** shows an application the user chooses from a list of countries and we then display a typical food from the selected country.

Example 16.8 Using *JComboBox* to Display Food Samplings

```java
/* Using JComboBox to show a sampling of international foods
 * Anderson, Franceschi
*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FoodSamplings extends JFrame
{
 private Container contents;
 private JComboBox<String> countries;
 private JLabel foodImage;

 private String [ ] countryList =
            { "France", "Greece", "Italy", "Japan", "USA" };
 private ImageIcon [ ] foods =
            {  new ImageIcon( "cheese.jpg" ),
               new ImageIcon( "fetaSalad.jpg" ),
               new ImageIcon( "pizza.jpg" ),
               new ImageIcon( "sushi.jpg" ),
               new ImageIcon( "hamburger.jpg" ) };

 public FoodSamplings( )
 {
  super( "Food samplings of various countries" );
  contents = getContentPane( );
  contents.setLayout( new FlowLayout( ) );

  // instantiate the components
  countries = new JComboBox<String>( countryList );
  countries.setMaximumRowCount( 4 );
  countries.setSelectedIndex( 2 );
  foodImage = new JLabel( foods[2] );

  // add components to the content pane
  contents.add( countries );
  contents.add( foodImage );

  // set up event handler
  ItemListenerHandler ilh = new ItemListenerHandler ( );
  countries.addItemListener( ilh );

  setSize( 350, 150 );
  setVisible( true );
 }

 private class ItemListenerHandler implements ItemListener
 {
  public void itemStateChanged( ItemEvent ie )
  {
   int index = countries.getSelectedIndex( );
   foodImage.setIcon( foods[index] );
  }
 }

 public static void main( String [ ] args )
 {
  FoodSamplings fs = new FoodSamplings( );
  fs.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

Description

We declare and instantiate the instance variable, *foods*, as an array of *ImageIcons* (lines 17–22). The *foods* array stores images of the food samplings in the same order as the countries in our list. The instance variable *countryList*, an array of *Strings* declared and initialized at lines 15–16, holds the names of the countries to be displayed in the list.

At line 31, we instantiate *countries*, our *JComboBox*, by passing to the constructor the *countryList* array. To set the maximum number of items that the user will see when the combo box opens, we call the *setMaximumRowCount* method with an argument of 4 (line 32). At line 33, we programmatically select the third item, *Italy*, using the method *setSelectedIndex*. In order to match the displayed food sampling with the combo box selection, we initialize the *foodImage* label with the third element of the *foods* array (line 34).

Notice that our list of countries is arranged alphabetically to simplify finding a desired country. In this case, we have only five countries; however, with a longer list, the alphabetical ordering makes it easy for the user to find a particular item in the list. Another option is to display the most likely choice first, then display the remaining items in alphabetical order. For example, if most users reside in the United States, you could list “USA” first in the list, with the remaining countries alphabetized starting in the second position of the list.

SOFTWARE ENGINEERING TIP

Arrange list items in a logical order so that the user can find the desired item quickly.

At lines 48–55, we define the *ItemListenerHandler* event handler. To determine which item the user has selected, we call the *getSelectedIndex* method (line 52). Using that returned index, we call the *setIcon* method of the *JLabel* class to match the image of the label to the selected country.

Figure 16.12 shows the application after the user has clicked on *“Greece”* inside the combo box. Try running the example, then modify the number of visible rows by changing the argument of the *setMaximumRowCount* method, and run the example again.

Figure 16.12 Running Example 16.8

Description

with these end-of-chapter questions

16.17.1	Multiple Choice Exercises Questions 5, 6, 7, 8
16.17.2	Reading and Understanding Code Questions 16, 19, 20, 21, 22, 23, 24, 25, 26
16.17.3	Fill In the Code Questions 28, 29, 31, 32
16.17.4	Identifying Errors in Code Questions 43, 44, 45
16.17.5	Debugging Area Questions 49, 50
16.17.6	Write a Short Program Question 51, 52, 53, 54, 55, 56, 57

CODE IN ACTION

On the companion website you will find a Flash movie with a step-by-step illustration of working with GUI components. Click on the link to start the movie.

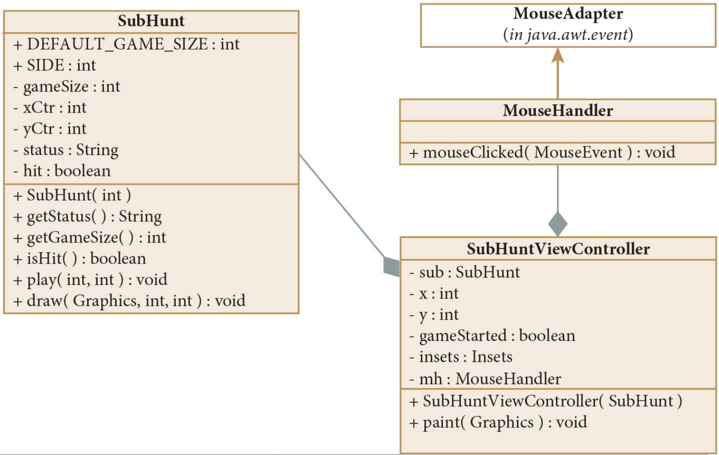
© Hemera Technologies/Photos.com/Thinkstock

16.10 Adapter Classes

We have learned that events are associated with event listener interfaces, and that we implement at least one interface in order to process an event. So far, we have used the *ActionListener* and *ItemListener* interfaces, each of which has only one method. For mouse events, there are two listeners: the *MouseListener* and the *MouseMotionListener*. The *MouseListener* interface specifies five methods to implement and the *MouseMotionListener* interface specifies two methods to implement.If we want to use *MouseListener* but need only one of its five methods to process a *MouseEvent*, we still have to implement the other four methods as “do-nothing” methods with empty method bodies. For convenience, Java provides **adapter classes**, each of which *implements* an interface and provides an empty body for each of the interface’s methods. For mouse events, the adapter classes are *MouseAdapter* and the *MouseMotionAdapter*. Thus, instead of implementing an interface, we can extend the appropriate adapter class and override only the method or methods we need. For example, if we want to process a *MouseEvent* using only one method of the five in the *MouseListener* interface, we can simply extend the *MouseAdapter* class and override that one method. We will do that in our next example.

SOFTWARE ENGINEERING TIP

When you need to implement only a few methods of a listener interface having more than one method, consider extending the corresponding adapter class instead.



SubHunt

+ DEFAULT_GAME_SIZE : int
+ SIDE : int
- gameSize : int
- xCtr : int
- yCtr : int
- status : String
- hit : boolean

+ SubHunt(int)
+ getStatus( ) : String
+ getGameSize( ) : int
+ isHit( ) : boolean
+ play(int, int) : void
+ draw(Graphics, int, int) : void

MouseAdapter
(*in java.awt.event*)

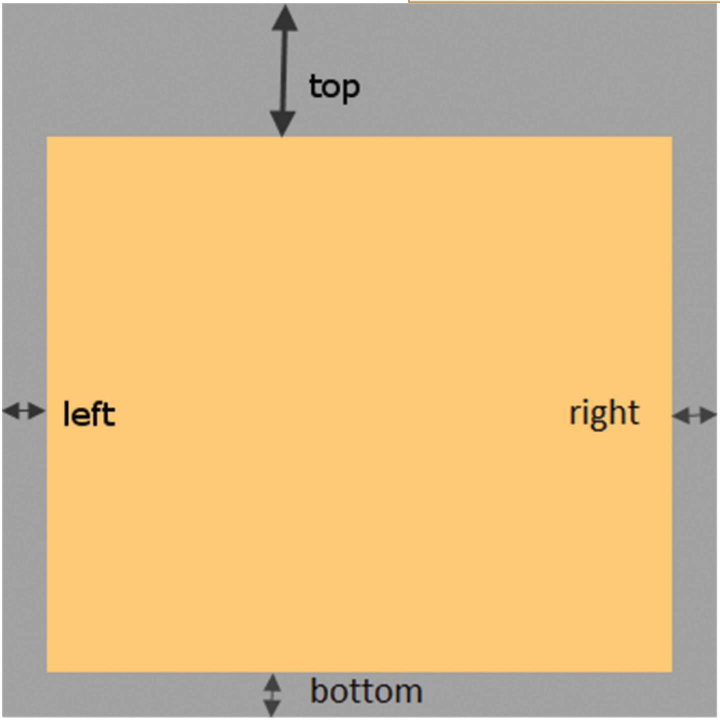
MouseHandler

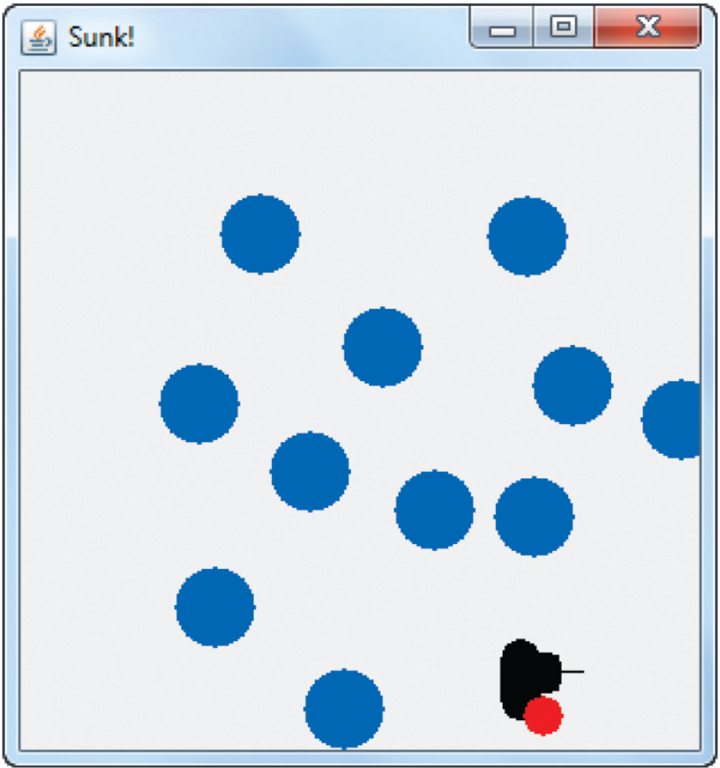
+ mouseClicked(MouseEvent) : void

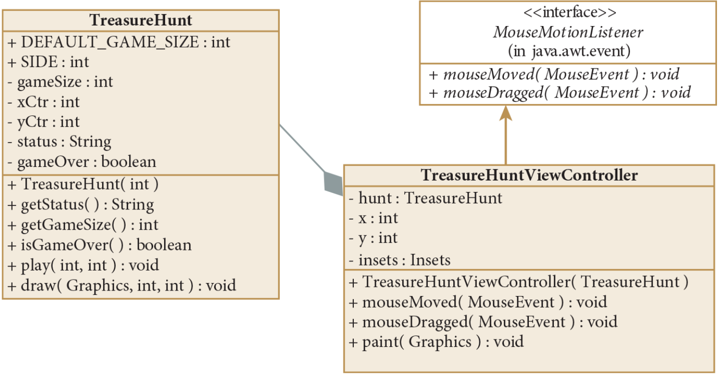
SubHuntViewController

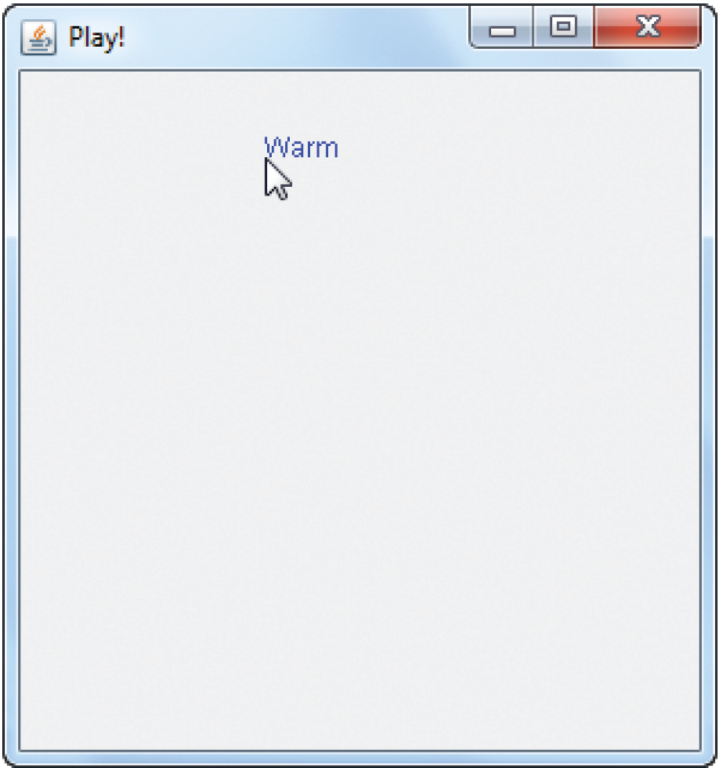
- sub : SubHunt
- x : int
- y : int
- gameStarted : boolean
- insets : Insets
- mh : MouseHandler

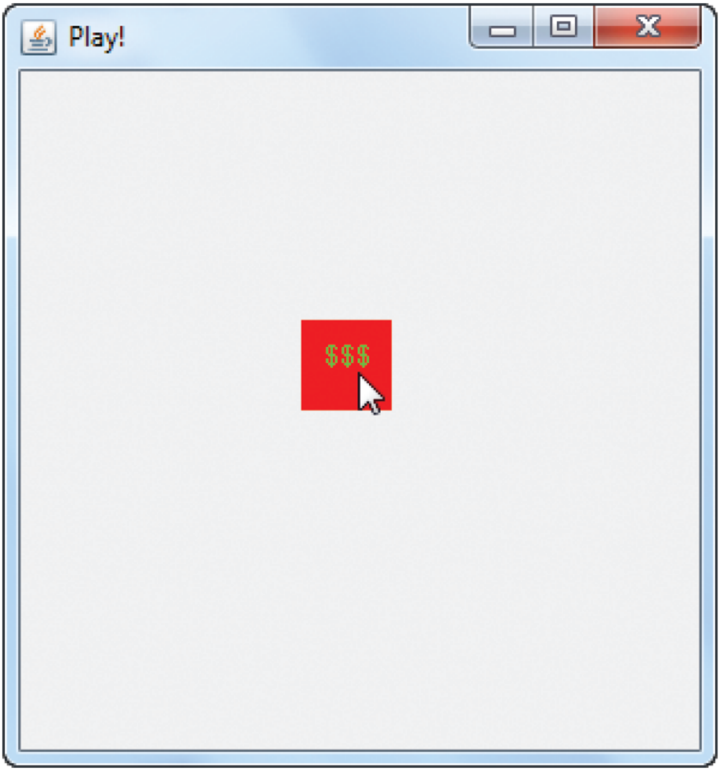
+ SubHuntViewController(SubHunt)
+ paint(Graphics) : void




Sunk!


TreasureHunt
+ DEFAULT_GAME_SIZE : int
+ SIDE : int
- gameSize : int
- xCtr : int
- yCtr : int
- status : String
- gameOver : boolean
+ TreasureHunt(int)
+ getStatus( ) : String
+ getGameSize( ) : int
+ isGameOver( ) : boolean
+ play(int, int) : void
+ draw(Graphics, int, int) : void
<<interface>>
MouseMotionListener
(in java.awt.event)
+ mouseMoved(MouseEvent) : void
+ mouseDragged(MouseEvent) : void
TreasureHuntViewController
- hunt : TreasureHunt
- x : int
- y : int
- insets : Insets
+ TreasureHuntViewController(TreasureHunt)
+ mouseMoved(MouseEvent) : void
+ mouseDragged(MouseEvent) : void
+ paint(Graphics) : void


Play!
Warm



16.11 Mouse Movements

A truly interactive application allows the user to point and click using the mouse. Any mouse activity (clicking, moving, or dragging) by the user generates a *MouseEvent*. When any mouse activity occurs, we will be interested in determining where on the window the user clicked, moved, or dragged the mouse. To determine the (*x, y*) coordinate of the mouse event, we can call two methods of the *MouseEvent* class, *getX* and *getY*, which are described in **Table 16.16**.

Useful Methods of the *MouseEvent* Class

Return Value	Method Name and Argument List

`int` `getX()`
returns the x value of the (*x,y*) coordinate of the mouse activity
`int` `getY()`
returns the y value of the (*x, y*) coordinate of the mouse activity

---

Description

Depending on the type of mouse activity, our application will implement either a *MouseListener* interface or a *MouseMotionListener* interface. As mentioned, both of the listener interfaces have been implemented as adapter classes that we can extend. The *MouseListener* interface includes the five *abstract* methods described in **Table 16.17**.

Methods of the *MouseListener* Interface

MouseListener Interface Method APIs

`public void mousePressed( MouseEvent e )`
called when the mouse button is pressed
`public void mouseReleased( MouseEvent e )`
called when the mouse button is released after being pressed
`public void mouseClicked( MouseEvent e )`
called when the mouse button is pressed and released
`public void mouseEntered( MouseEvent e )`
called when the mouse enters the registered component
`public void mouseExited( MouseEvent e )`
called when the mouse exits the registered component

Description

To illustrate how to use the *MouseAdapter* class, we will build a simple submarine hunt game. A submarine is hidden somewhere in the window, and the user will try to sink the submarine by clicking the mouse at various locations in the window, simulating the dropping of a depth charge. Each time the user clicks the mouse, we will indicate how close that click is to the submarine. If the user clicks too far from the submarine, we will display “In the water” in the title bar and draw a blue circle where the mouse was clicked. If the user clicks close to the submarine, we will display “Close …” in the title bar. Finally, if the submarine is hit, we will change the title bar to “Sunk!”, display the submarine, and remove the listener so that the game ends.

In this application, we want to design our classes so that they are as reusable as possible. To this end, we will design our classes using the **Model- View-Controller** architecture.

The **Model** encapsulates the application's functionality: It should be designed so that it can be reused in a different context, in particular with a different user interface.

The **View** encapsulates the user interface of the application: It should also be designed so that it can be reused with a different Model.

The **Controller** is the middleman between the View and the Model. When the user interacts with the View, the Controller asks the Model to perform some calculations, and then updates the View accordingly. If possible, the Controller should also be designed so that it is independent from the Model and the View so that it can be reused with other Views and other Models.

As much as possible, it is good design practice to separate the Model, the View, and the Controller. It makes the application components easier to design, code, maintain, modify, and reuse.

Figure 16.13 shows the UML diagram for this application. We first code our Model, the *SubHunt* class (**Example 16.9**). It encapsulates the submarine hunt game and enables us to create a game of a given size, enable play, and enforce the rules of the game.

Figure 16.13 UML Diagram for *SubHunt* and *SubHuntViewController*

Description

Example 16.9 The *SubHunt* Class

```
/* SubHunt class
*  Anderson, Franceschi
*/

import java.awt.Graphics;
import java.awt.Color;
import java.util.Random;

public class SubHunt
{
 public static int DEFAULT_GAME_SIZE = 300;
 public static int SIDE = 36; // size of submarine

 private int gameSize;
 private int xCtr; // x coordinate of center of submarine
 private int yCtr; // y coordinate of center of submarine
 private String status = "";
 private boolean hit;

 /** Constructor
 * @param newGameSize game size
 */
 public SubHunt( int newGameSize )
 {
  if ( newGameSize > SIDE )
    gameSize = newGameSize;
  else
    gameSize = DEFAULT_GAME_SIZE;
  // generate submarine center
  Random random = new Random( );
  xCtr = SIDE / 2 + random.nextInt( gameSize - SIDE );
  yCtr = SIDE / 2 + random.nextInt( gameSize - SIDE );
  hit = false;
 }

 /** getStatus
 * @return status
 */
 public String getStatus( )
 {
  return status;
 }

 /** getGameSize
 * @return gameSize
 */
 public int getGameSize( )
 {
  return gameSize;
 }

 /** isHit
 * @return hit
 */
 public boolean isHit( )
 {
  return hit;
 }

 /** play
 * @param x the x coordinate of the play
 * @param y the y coordinate of the play
 */
 public void play( int x, int y )
 {
  // is click within the submarine?
  if ( Math.abs( x - xCtr ) < SIDE / 2
       && Math.abs( y - yCtr ) < SIDE / 2 )
  {
    status = "Sunk!";
    hit = true;
  }
  // is click close?
  else if ( Math.abs( x - xCtr ) < 2 * SIDE
            && Math.abs( y - yCtr ) < 2 * SIDE )
    status = "Close ...";
  // click is too far from submarine
  else
    status = "In the water";
 }

 /** draw
 * @param g a Graphics object
 * @param x the x coordinate of the play
 * @param y the y coordinate of the play
 */
 public void draw( Graphics g, int x, int y )
 {
  if ( status.equals( "Sunk!" ) )
  {
    // draw sunken submarine
    g.setColor( Color.BLACK );
    g.fillRoundRect( xCtr - SIDE/2, yCtr - SIDE/2,
                     SIDE/2, SIDE, SIDE/2, SIDE/2 );

    g.fillRoundRect( xCtr - SIDE/4, yCtr - SIDE/3,
                     SIDE/2, SIDE/2, SIDE/4, SIDE/4 );

    g.drawLine( xCtr + SIDE/4, yCtr - SIDE/9,
                xCtr + SIDE/2, yCtr - SIDE/9 );

    // draw red depth charge
    g.setColor( Color.RED );
    g.fillOval( x - SIDE/4, y - SIDE/4, SIDE/2, SIDE/2 );
  }
  else if ( status.equals( "In the water" ) ) // draw blue circle
  {
    g.setColor( Color.BLUE );
    g.fillOval( x - SIDE/2, y - SIDE/2, SIDE, SIDE );
  }
  // else Close .    .    . , do not draw
 }
}
```

Description

Example 16.10 shows the *SubHuntViewController* class. It combines the View and the Controller. Later in the chapter, we will show you how to separate the View and the Controller in order to make the View reusable.

Example 16.10 The *SubHuntViewController* Class

```java
/* SubHuntViewController class
*  Anderson, Franceschi
*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SubHuntViewController extends JFrame
{
 private int x; // current x mouse position
 private int y; // current y mouse position
 private SubHunt sub; // submarine
 private boolean gameStarted = false;
 private Insets insets;
 private MouseHandler mh; // mouse event handler

 public SubHuntViewController( SubHunt newSub )
 {
  super( "Play!" );

  sub = newSub;
  // instantiate event handler and register listener on window
  mh = new MouseHandler( );
  addMouseListener( mh );

  // size window to include insets
  setVisible( true );
  insets = getInsets( );
  setSize( sub.getGameSize( ) + insets.left + insets.right,
           sub.getGameSize( ) + insets.top + insets.bottom );
 }

 private class MouseHandler extends MouseAdapter
 {
  public void mouseClicked( MouseEvent me )
  {
   x = me.getX( ) - insets.left;
   y = me.getY( ) - insets.top;
   sub.play( x, y );
   setTitle( sub.getStatus( ) );
   repaint( );
  }
 }

 public void paint( Graphics g )
 {
  if ( !gameStarted )
  {
   super.paint( g );
   gameStarted = true;
  }
  g.translate( insets.left, insets.top );
  sub.draw( g, x, y );
  if ( sub.isHit( ) )
    removeMouseListener( mh );
 }
}
```

Description

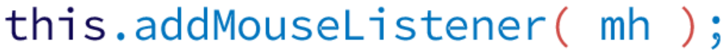
The constructor (lines 18–32) defines and displays the original View. The *MouseHandler* private class (lines 34–44) and the *paint* method (lines 46–57) make up the Controller. The Model is represented by the instance variable *sub* (line 13). When the user interacts with the View by clicking on the window, the Controller captures that information, calls the appropriate method of the *SubHunt* class, and updates the View accordingly.

In this game, the only mouse action we care about is a click; therefore, we are interested in only one method of the *MouseListener* interface: *mouseClicked*. To simplify our code, our handler extends the *MouseAdapter* class, which provides do-nothing implementations for the five *MouseListener* methods, so our event handler needs to override only the *mouseClicked* method.

At lines 11–16, we declare our instance variables. The *ints x* and *y* represent the *x* and *y* coordinates where the player clicked the mouse relative to the entire window. The *SubHunt* instance variable *sub* represents the submarine hunt game that we display inside the window. The constructor receives a *SubHunt* object as a parameter, which it stores in the *sub* instance variable on line 22. When the user plays, we use the *sub* reference to call the various methods of the *SubHunt* class to enable play and enforce the rules of the game.

In this example, we have set up the listener reference, *mh*, as an instance variable (line 16) rather than the reference as a local variable to the constructor, as we have done in previous examples. We define *mh* as an instance variable because, when the game is over, the event handler needs to access that listener to turn it off. The event handler is instantiated at line 24.

We want the listener to handle mouse clicks anywhere in the window, so we register the *mouseListener* on our window (*JFrame*) component, which is the *SubHuntViewController* object. Thus, we do not use an object reference to call the *addMouseListener* method, and line 25 is equivalent to:



```
this.addMouseListener( mh );
```

Description

When the user clicks inside a *JFrame window*, the *x* and *y* coordinates returned by the *mouseClicked* method are relative to the top-left corner of the window; that is, (0,0) is the top-left corner of the window. However, for a *JFrame*, the title bar is in the top-left corner. In fact, a *JFrame* has borders at the top (where the title bar is displayed), the left, the right, and the bottom. These four areas together are called the **insets** of the frame. Thus, in coding the sub hunt game, we need to adjust the *x* and *y* coordinates so that the game is played inside a rectangle that excludes the insets.

Figure 16.14 shows the *JFrame* and sub hunt game rectangles in gray and orange, respectively.

Figure 16.14 The Game Rectangle within the *JFrame*

Description

The *Insets* class (in the *java.awt* package) allows us to determine the sizes of the borders. The *getInsets* method of the *Container* class, inherited by *JFrame* and shown in **Table 16.18**, returns the *Insets* object for the current frame. The *Insets* class includes four *public* instance variables shown in **Table 16.19**, which we can access directly in order to retrieve the sizes of the four borders of the frame.

The *getInsets* Method Inherited from the *Container* Class

Method	Description
`Insets getInsets( )`	Returns an object representing the insets of this *Container*

Description

The Instance Variables of the *Insets* Class

Instance Variables	Description
`public int top`	The size of the top border
`public int left`	The size of the left border
`public int bottom`	The size of the bottom border
`public int right`	The size of the right border

Description

At lines 28–30, we retrieve the insets for the current *JFrame* and size the window so that its inside rectangle matches the dimensions of the rectangle of the *SubHunt* instance variable *sub*.

At lines 34–44, we define our mouse event handler, *MouseHandler*, as extending *MouseAdapter*. Inside the method *mouseClicked*, overridden at lines 36–43, we call the *MouseEvent* methods *getX* and *getY* to get the (*x, y*) coordinate where the mouse was clicked relative to the entire window. We adjust these values by the sizes of the insets before storing them in the *x* and *y* instance variables. Then we call the *play* method of *SubHunt* with *sub* (line 40) to process the play, passing the position of the click. Next, we update the title of the window by getting the updated status, and display that status in the window title bar by calling the *setTitle* method (line 41). Finally, we call the *repaint* method (line 42) to update the display. The *repaint* method is inherited from the *Component* class and has the API shown in **Table 16.20**. Since we cannot call *paint* directly, we call *repaint to* force a call to the *paint* method in order to update the drawing in the window.

The *repaint* Method API in the *Component* Class

Return Value	Method Name and Argument List
`void`	`repaint( )`
automatically forces a call to the *paint* method	

Description

For this game, we want to preserve any previously drawn shapes. Thus, in the *paint* method (lines 46–57), rather than clear the window and redraw every time, we set our flag *gameStarted* to *false* when the *game* starts (line 14). This causes us to call *super.paint* to clear the window the first time *paint* is called (lines 48–52) and then set *gameStarted* to *true* so that we skip any further calls to *super.paint*.

In order to draw the result of the last click, we first call the *translate* method of the *Graphics* class (line 53), passing the left and top insets to adjust the click position of the coordinate to the top-left corner of the sub hunt rectangle. The *translate* method's API is shown in **Table 16.21**. We then call the *SubHunt draw* method, passing the *Graphics* object *g* and the mouse position.

The *translate* Method of the *Graphics* Class

Method	Description
`void translate( int x, int y )`	translates the origin of the graphics context to the point (*x, y*) in the current coordinate system

Description

Finally, if the game is over, we remove *mh* as a listener to this object (line 56) by calling the *removeMouseListener* method, inherited from the *Component* class.

The *PlaySubHunt* class, shown in **Example 16.11**, includes the *main* method to create a *SubHunt* object and a *SubHuntViewController* application. **Figure 16.15** shows a run of this game. At this point, the user has sunk the submarine.

Example 16.11 The *PlaySubHunt* Class

```
/* PlaySubHunt class
*  Anderson, Franceschi
*/

import javax.swing.JFrame;

public class PlaySubHunt
{
 public static void main( String [ ] args )
 {
  SubHunt sub = new SubHunt( SubHunt.DEFAULT_GAME_SIZE );
  SubHuntViewController subGui = new SubHuntViewController( sub );
  subGui.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

Description

Figure 16.15 A Run of the Sub Hunt Game

Description

The second interface related to mouse events is *MouseMotionListener*, which has two methods, *mouseMoved* and *mouseDragged*, shown in **Table 16.22**. Dragging the mouse is defined as the user moving the mouse with the mouse button pressed, but not released.

MouseMotionListener Interface Method APIs

MouseMotionListener **Interface Method APIs**

`public void mouseMoved( MouseEvent e )`

called when the mouse is moved onto a component

`public void mouseDragged( MouseEvent e )`

called when the mouse button is pressed on a component and the mouse is dragged

---

Description

In order to illustrate how to use the *MouseMotionListener* interface with a *MouseEvent* class, we will build a simple treasure hunt game that is similar to the submarine hunt game. A treasure is hidden somewhere in the window, and the user will try to find it, not by clicking the mouse button, but by moving the mouse inside the window. Depending on how close the mouse is to the treasure, we will display a message at the mouse location so that the user can find the treasure and win the game.

Figure 16.16 shows the UML diagram for the Treasure Hunt Game.

Figure 16.16 UML Diagram for the Treasure Hunt Game

Description

Again, we first code the Model, the class to encapsulate the functionality of our treasure hunt game (**Example 16.12**). It enables us to create a game of a given size, handle plays, determine when the game is over, and display the status of the game.

Example 16.12 The *TreasureHunt* Class

```java
/* TreasureHunt class
*  Anderson, Franceschi
*/

import java.awt.*;
import java.util.Random;

public class TreasureHunt
{
 public static int DEFAULT_GAME_SIZE = 300; // side of window
 public static int SIDE = 40; // side of treasure

 private int gameSize;
 private int xCtr; // x coordinate of center of treasure
 private int yCtr; // y coordinate of center of treasure
 private String status = ""; // message
 private boolean gameOver = false;

 /** Constructor
 */
 public TreasureHunt( int newGameSize )
 {
  if ( newGameSize > SIDE )
    gameSize = newGameSize;
  else
    gameSize = DEFAULT_GAME_SIZE;
  // generate treasure center
  Random random = new Random( );
  xCtr = SIDE / 2 + random.nextInt( gameSize - SIDE );
  yCtr = SIDE / 2 + random.nextInt( gameSize - SIDE );
 }

 /** getGameSize
 * @return gameSize
 */
 public int getGameSize( )
 {
  return gameSize;
 }

 /** isGameOver
 * @return gameOver
 */
 public boolean isGameOver( )
 {
  return gameOver;
 }

 /** play
 * @param x the x coordinate of the play
 * @param y the y coordinate of the play
 */
 public void play( int x, int y )
 {
  // is mouse within treasure?
  if ( Math.abs( x - xCtr ) < SIDE / 2
       && Math.abs( y - yCtr ) < SIDE / 2 )
  {
    gameOver = true;
    status = "Found";
  }
  // is mouse within half-length of the treasure?
  else if ( Math.abs( x - xCtr ) < ( 1.5 * SIDE )
            && Math.abs( y - yCtr ) < ( 1.5 * SIDE ) )
    status = "Hot";
  // is mouse within 1 length of the treasure?
  else if ( Math.abs( x - xCtr ) < ( 2 * SIDE )
            && Math.abs( y - yCtr ) < ( 2 * SIDE ) )
    status = "Warm";
  // is mouse within 2 lengths of the treasure?
  else if ( Math.abs( x - xCtr ) < ( 3 * SIDE )
            && Math.abs( y - yCtr ) < ( 3 * SIDE ) )
    status = "Lukewarm";
  else // mouse is not near treasure
    status = "Cold";
 }

 /** draw
 * @param g a Graphics object
 * @param x the x coordinate of the play
 * @param y the y coordinate of the play
 */
 public void draw( Graphics g, int x, int y )
 {
  g.setColor( Color.BLUE );

  if ( status.equals( "Found" ) ) // if found, draw treasure
  {
   g.setColor( Color.RED );
   g.fillRect( xCtr - SIDE / 2, yCtr - SIDE / 2, SIDE, SIDE );
   g.setColor( Color.GREEN );
   g.drawString( "$$$", xCtr - SIDE / 4, yCtr );
  }
  else
   g.drawString( status, x, y ); // display current status
 }
}
```

Description

The *status* (line 16) instance variable, a *String*, stores the state of the player's position compared to the location of the treasure. We also define a *boolean* flag variable, *gameOver*, which is initially *false* (line 17); when the user finds the treasure, we will change it to *true.*

In the constructor, we randomly generate the (*x*, *y*) coordinate for the center of the treasure and store the generated values in the *xCtr* and *yCtr* instance variables (lines 27–30).

The *play* method (lines 49–76) sets the value of *status*. If the player is far from the treasure, we set the value of *status* to "Cold." As the user moves closer and closer to the treasure, the value becomes "Lukewarm," then "Warm," and then "Hot," and finally "Found" when the player finds the treasure. If the treasure is found, we set the value of *gameOver* to *true* (line 59).

Like the sub hunt game, the *draw* method (lines 78–96) draws what is happening in the game. We first test if the value of *status* is "Found" (line 87). If it is, the treasure has been found and we reveal its location by drawing it (lines 89–92). If the treasure has not been found, we display the *String status* at the current mouse location (line 95), determined by the parameters *x* and *y* sent to the *draw* method.

Example 16.13 shows the *TreasureHuntViewController* class. Like the sub hunt example, it combines the View and the Controller. The constructor (lines 17–32) defines and displays the original View. The *mouseMoved*, *mouseDragged*, and *paint* methods (lines 34–43, 45–46, and 48–53, respectively) make up the Controller. The Model is represented by the instance variable *hunt* (line 14), where we store the *TreasureHunt* object sent to the constructor as a parameter (line 17).

Example 16.13 The *TreasureHuntViewController* Class

```
/* TreasureHuntViewController class
*  Anderson, Franceschi
*/

import javax.swing.JFrame;
import java.awt.*;
import java.awt.event.*;

public class TreasureHuntViewController extends JFrame
             implements MouseMotionListener
{
  private int x;
  private int y;
  private TreasureHunt hunt;
  private Insets insets;

  public TreasureHuntViewController( TreasureHunt newHunt )
  {
   super( "Play!" );

   hunt = newHunt;
   // application registers on itself
   // since it is a MouseMotionListener itself
   addMouseMotionListener( this );
   setSize( hunt.getGameSize( ), hunt.getGameSize( ) );
   setVisible( true );

   // resize window to include insets
   insets = getInsets( );
   setSize( getWidth( ) + insets.left + insets.right,
            getHeight( ) + insets.top + insets.bottom );
  }

  public void mouseMoved( MouseEvent me )
  {
   // get location of mouse
   x = me.getX( ) - insets.left;
   y = me.getY( ) - insets.top;
   hunt.play( x, y );
   repaint( );
   if ( hunt.isGameOver( ) )
     removeMouseMotionListener( this );
  }

  public void mouseDragged( MouseEvent me )
  { } // we do not want to process mouse drag events

  public void paint( Graphics g )
  {
   super.paint( g );
   g.translate( insets.left, insets.top );
   hunt.draw( g, x , y );
  }
}
```

Description

In this application, instead of coding the event handler as a *private* inner class, we define our application class as implementing the *MouseMotionListener* interface. As a result, our application is a listener, and we register the listener on itself. Thus, in our class definition, we include the clause *implements MouseMotionListener* (lines 9–10). Again, the instance variables *x* and *y* (lines 12–13) will store the current location of the mouse.

At lines 22–24, we register this *TreasureHuntViewController* object on itself as a *MouseMotionListener*. In this game, the only event we want to handle is the user moving the mouse. Because *TreasureHuntViewController* implements the *MouseMotionListener* interface, however, we must implement both the *mouseMoved* and the *mouseDragged* methods. Thus, we provide a method body for the *mouseMoved* method, and we implement *mouseDragged* with an empty body (lines 45–46). Note that we could not use the *MouseMotionAdapter* class because our application (which is the listener) already extends the *JFrame* class. Remember that a class can extend only one class, but can implement multiple interfaces.

The *mouseMoved* method is implemented at lines 34–43. At lines 36–38, we assign the (*x*, *y*) coordinate of the mouse position within the playing rectangle to the instance variables *x* and *y*, using the *getX* and *getY* methods of the *MouseEvent* class and by the sizes of the *left* and *top* insets. Then we call the *play* method of *TreasureHunt* with *hunt* (line 39) to process the play and call the *repaint* method (line 40) to update the window. If the game is over (i.e., the treasure has been found), we remove this object, the current *JFrame*, as a listener (line 42) by calling the *removeMouseMotionListener* method.

We begin the *paint* method (lines 48–53) with a call to the *paint* method of the *JFrame* superclass to clear the contents of the window. At line 51, we call the *translate* method with the *Graphics* object *g* so that its (*0*, *0*) coordinate is at the top left corner of the game rectangle. Then we draw the result of the current move (line 52) by calling the *draw* method from *TreasureHunt* with *hunt*, passing the *Graphics* object *g* and the mouse location within the playing rectangle.

Finally, the *PlayTreasureHunt* class, shown in **Example 16.14**, includes the *main* method to create a

TreasureHuntViewController application showing a *TreasureHunt* game. **Figures 16.17** and **16.18** show the program running. In Figure 16.17, the user is getting close to the treasure, and in Figure 16.18, the user has found the treasure.

Example 16.14 The *PlayTreasureHunt* Class

```
/* PlayTreasureHunt class
*  Anderson, Franceschi
*/

import javax.swing.JFrame;

public class PlayTreasureHunt
{
 public static void main( String [ ] args )
 {
  TreasureHunt th = new TreasureHunt(
                         TreasureHunt.DEFAULT_GAME_SIZE );
  TreasureHuntViewController huntGui
    = new TreasureHuntViewController( th );
  huntGui.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

Description

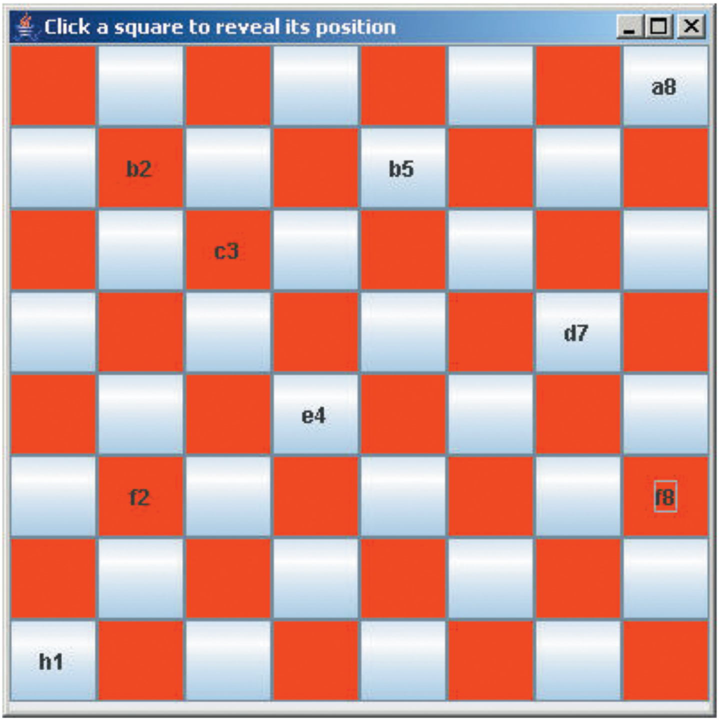
Figure 16.17 The User is Getting Close to the Treasure

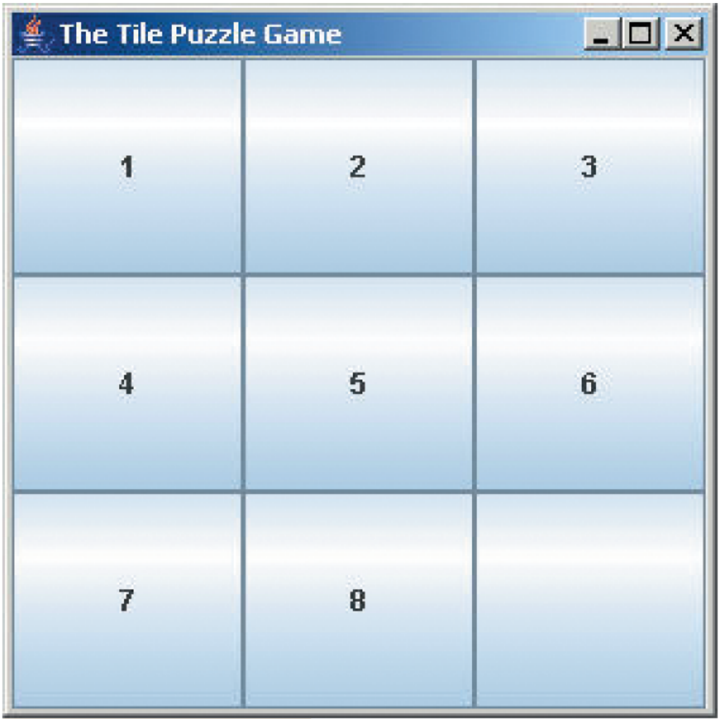
Description

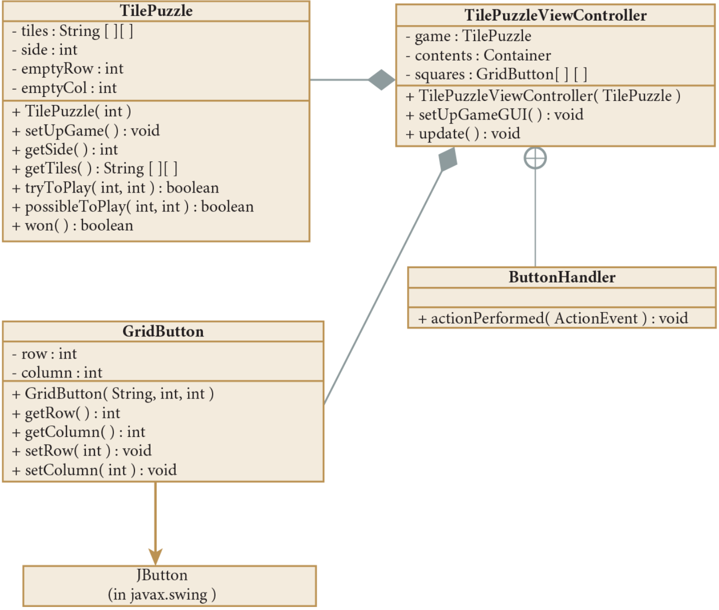
Figure 16.18 The User Has Found the Treasure

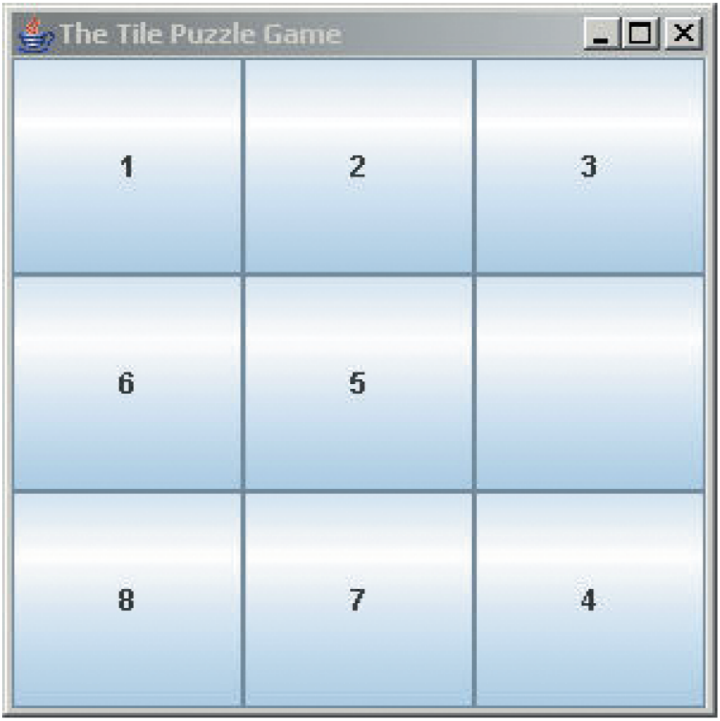
Description

In both the sub hunt and treasure hunt examples, the Model is reusable in another application. However, the View and the Controller are combined in the same class, and are not reusable. Later in the chapter, we will make the View reusable too.


Click a square to reveal its position
a8
b2
b5
c3
d7
e4
f2
f8
h1


The Tile Puzzle Game
1
2
3
4
5
6
7
8


TilePuzzle
- tiles : String [ ][ ]
- side : int
- emptyRow : int
- emptyCol : int
+ TilePuzzle(int)
+ setUpGame( ) : void
+ getSide( ) : int
+ getTiles( ) : String [ ][ ]
+ tryToPlay(int, int) : boolean
+ possibleToPlay(int, int) : boolean
+ won( ) : boolean
TilePuzzleViewController
- game : TilePuzzle
- contents : Container
- squares : GridButton[ ] [ ]
+ TilePuzzleViewController(TilePuzzle)
+ setUpGameGUI( ) : void
+ update( ) : void
ButtonHandler
+ actionPerformed(ActionEvent) : void
GridButton
- row : int
- column : int
+ GridButton(String, int, int)
+ getRow( ) : int
+ getColumn( ) : int
+ setRow(int) : void
+ setColumn(int) : void
JButton
(in javax.swing)



16.12 Layout Managers: *GridLayout*

We have been using the *FlowLayout* layout manager because it is the easiest to use. The *FlowLayout* layout manager adds new components from left to right, creating as many rows as needed. A problem, however, is that if the user resizes the window, the components are rearranged to fit the new window size. Typically, GUIs can be quite complex and use many components, all of which need to be organized so that the user interface is effective and easy to use even if the user resizes the window. Two useful layout managers that give us more control over the organization of components within a window are the *GridLayout* and the *BorderLayout*.

Regardless of the layout manager used, the constructor of our application will still perform the following setup operations:

- declare and instantiate the layout manager
- use the *setLayout* method of the *Container* class to set the

layout manager of the content pane of the *JFrame*
- instantiate components
- add components to the content pane using one of the *add* methods of the *Container* class

The *GridLayout* organizes the container as a grid. We can visualize the layout as a table made up of equally sized cells in rows and columns. Each cell can contain one component. The first component added to the container is placed in the first column of the first row; the second component is placed in the second column of the first row, and so on. When all the cells in a row are filled, the next component added is placed in the first cell of the next row.

Table 16.23 shows two constructors of the *GridLayout* class.

GridLayout Constructors

<table>
<tr><th colspan="2">Constructors</th></tr>
<tr><th>Class</th><th>Constructor</th></tr>
<tr><td><code>GridLayout</code></td><td><code>GridLayout(int numberOfRows, int numberOfColumns)</code></td></tr>
<tr><td colspan="2">creates a grid layout with the number of rows and columns specified by the arguments.</td></tr>
<tr><td><code>GridLayout</code></td><td><code>GridLayout(int numberOfRows, int numberOfColumns, int hGap, int vGap)</code></td></tr>
<tr><td colspan="2">creates a grid layout with the specified number of rows and columns and with a horizontal gap of <i>hGap</i> pixels between columns and a vertical gap of <i>vGap</i> pixels between rows. Horizontal gaps are also placed at the left and right edges, and vertical gaps are placed at the top and bottom edges.</td></tr>
</table>

Description

In order to focus on how to use a *GridLayout*, we keep the next example as simple as possible and do not use the MVC architecture. **Example 16.15** shows how to use a *GridLayout* to display a chessboard. Each square's position on a chessboard can be identified by a letter and a number. From the standpoint of the white player, the lower right square is h1, and from the standpoint of the black player, the lower right square is a8. When the user clicks on a square on the chessboard, our application displays its position. We show the chessboard oriented as if the white player is on the left and the black player is on the right.

Example 16.15 The *ChessBoard* Class

```
/* Using GridLayout to organize our window
   Anderson, Franceschi
*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ChessBoard extends JFrame
{
 private Container contents;
 private JButton [ ][ ] squares;
 private String [ ] letters =
    { "a", "b", "c", "d", "e", "f", "g", "h" };
 private int [ ] digits = { 1, 2, 3, 4, 5, 6, 7, 8 };

 public ChessBoard( )
 {
  super( "Click a square to reveal its position" );
  contents = getContentPane( );

  // set layout to an 8-by-8 Grid
  contents.setLayout( new GridLayout( letters.length, digits.length ) );

  squares = new JButton[letters.length][digits.length];

  ButtonHandler bh = new ButtonHandler( );

  for ( int i = 0; i < letters.length; i++ )
  {
   for ( int j = 0; j < digits.length; j++ )
   {
    // instantiate JButton array
    squares[i][j] = new JButton( );

    // make every other square red
    if ( ( i + j ) % 2 == 0 )
      squares[i][j].setBackground( Color.RED );

    // add the JButton
    contents.add( squares[i][j] );

    // register listener on button
    squares[i][j].addActionListener( bh );
   }
  }

  setSize( 400, 400 );
  setVisible( true );
 }

 private class ButtonHandler implements ActionListener
 {
  public void actionPerformed( ActionEvent ae )
  {
    for ( int i = 0; i < letters.length; i++ )
    {
      for ( int j = 0; j < digits.length; j++ )
      {
        if ( ae.getSource( ) == squares[i][j] )
        {
          squares[i][j].setText( letters[i] + digits[j] );
          return;
        }
      }
    }
  }
 }

 public static void main( String [ ] args )
 {
   ChessBoard myGame = new ChessBoard( );
   myGame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

Description

A two-dimensional array of *JButtons*, named *squares* (line 12), make up the chessboard. We use two arrays, one of *Strings*, named *letters*, and one of *ints*, named *digits*, to make up the position of each square (lines 13–15). The position of a square is composed of a letter representing the row (a–h) and a number representing the column (1–8). When the user clicks on a button of the chessboard, we set the text of that button to display its position on the board, composing the name from the data in the arrays *letters* and *digits*.

At line 23, we instantiate a *GridLayout* anonymous object using the constructor with two arguments representing the number of rows and columns in the grid, here the number of elements of the *letters* array and the number of elements of the *digits* array. Still at line 23, we pass the *GridLayout* anonymous object as the argument of the *setLayout* method. Thus, components will be added cell-by-cell to an 8-by-8 grid, with each row being filled before starting the next row.

At line 27, we declare and instantiate the listener *bh*, a *ButtonHandler* object reference. Since we are interested in events related to buttons, our *Button-Handler private* inner class *implements* the *ActionListener* interface and overrides the *actionPerformed* method. At line 25, we instantiate the two-dimensional array *squares*. Because we have a two-dimensional array of *JButtons*, we use nested *for* loops at lines 29–46 to instantiate the *JButtons*, add them to the content pane, and register the listener on all of the buttons. Inside the inner loop, the value of the expression *(i + j)* will alternate between an even and odd number. Accordingly, at lines 36–38, we set the background color of every other button to red.

In the *actionPerformed* method, we use nested *for* loops at lines 56–66 to identify the source of the event, that is, which button the user clicked. We then set the text of that button to its board position (line 62), using the corresponding elements in the two arrays *letters* and *digits*. Having found the source of the event, we then exit the event handler via the *return* statement (line 63) to interrupt the *for* loops, and thus avoid unnecessary processing.

Figure 16.19 shows the example running. When you run this program, click on various squares; resize the window, and check that the layout manager maintains an 8-by-8 grid.

Figure 16.19 Running Example 16.15

Description

Layout managers can be set dynamically, based on user input. For example, the user could enter the number of rows or columns of the grid. Or based on user input, we can rearrange the components using another layout manager, such as *FlowLayout*. The user could also instruct us to remove components and add others. Our next example, the Tile Puzzle game, will illustrate some of these capabilities.

In the Tile Puzzle game, eight tiles displaying the digits 1 through 8 are scrambled on a 3-by-3 grid, leaving one cell empty. Any tile adjacent to the empty cell can be moved to the empty cell by clicking on the numbered tile. The goal is to rearrange the tiles so that the numbers are in the correct order, as shown in **Figure 16.20**.

Figure 16.20 The Winning Position of a 3-by-3 Tile Puzzle Game

Description

The Tile Puzzle game can also be played on a 4-by-4, a 5-by-5, and more generally, an *n*-by-*n* grid. We will set up a 3-by-3 grid for the first game, then will randomly select a 3-by-3, a 4-by-4, a 5-by-5, or a 6-by-6 grid for subsequent games. Once you understand this example, you can modify it to allow the user to specify the size of the grid.

Before designing the GUI class, we first code the Model, a class to encapsulate the functionality of the tile puzzle game; the *TilePuzzle* class (**Example 16.16**) handles the creation of a game of a given size, enables play, and enforces the rules of the game.

Example 16.16 The *TilePuzzle* Class

```java
/* TilePuzzle class
*  Anderson, Franceschi
*/

public class TilePuzzle
{
 private String [ ][ ] tiles;
 private int side; // grid size
 private int emptyRow;
 private int emptyCol;

 /** constructor
 *  @param  newSide  grid size
 */
 public TilePuzzle( int newSide )
 {
  setUpGame( newSide );
 }

 /** setUpGame
 *  @param  newSide   grid size
 */
 public void setUpGame( int newSide )
 {
    if ( newSide < 1 )
      side = 3;
    else
      side = newSide;
    emptyRow = side - 1;
    emptyCol = side - 1;
    tiles = new String[side][side];

  // initialize tiles
  for ( int i = 0; i < side; i++ )
  {
   for ( int j = 0; j < side; j++ )
   {
     tiles[i][j] = String.valueOf( ( side * side )
                            - ( side * i + j + 1 ) );
   }
  }
  // set empty cell label to blank
  tiles[side - 1][side - 1] = "";
 }

 /** getSide
 *  @return side
 */
 public int getSide( )
 {
   return side;
 }

 /** getTiles
 *  @return a copy of tiles
 */
 public String [ ][ ]  getTiles( )
 {
  String[ ][ ] copyOfTiles = new String[side][side] ;

  for ( int i = 0; i < side; i++ )
  {
   for ( int j = 0; j < side; j++ )
   {
     copyOfTiles[i][j] = tiles[i][j];
   }
  }
  return copyOfTiles;
 }

 /** tryToPlay
 *  enable play if play is legal
 *  @return true if the play is legal, false otherwise
 */
 public boolean tryToPlay( int row, int col )
 {
  if ( possibleToPlay( row, col ) )
  {
   // play: switch empty String and tile label at row, col
   tiles[emptyRow][emptyCol] = tiles[row][col];
   tiles[row][col] =  "";
   // update emptyRow and emptyCol
   emptyRow = row;
   emptyCol = col;
   return true;
  }
  else
   return false;
 }

 /** possibleToPlay
 *  @return true if the play is legal, false otherwise
 */
 public boolean possibleToPlay( int row, int col )
 {
  if ( ( col == emptyCol && Math.abs( row - emptyRow ) == 1 )
         || ( row == emptyRow && Math.abs( col - emptyCol ) == 1 ) )
    return true;
  else
    return false;
 }

 /** won
 *  @return true if the tiles are all at the right place, false otherwise
 */
 public boolean won( )
 {
  for ( int i = 0; i < side ; i++ )
  {
   for ( int j = 0; j < side; j++ )
   {
     if ( !( tiles[i][j].equals(
                       String.valueOf( i * side + j + 1 ) ) )
                       && ( i != side - 1 || j != side - 1 )  )
       return false;
   }
  }
  return true;
 }
}
```

Description

The instance variable *tiles* (line 7), a two-dimensional array of *Strings*, stores the state of the puzzle. Each element of *tiles* will be a cell in the puzzle grid. The instance variable *side*, declared at line 8, represents the size of the grid. The instance variables *emptyRow* and *emptyCol*, declared at lines 9–10, identify the empty cell in the puzzle grid.

The constructor (lines 12–18) calls the *setUpGame* method (lines 20–44), passing the size of the grid (*newSide*) as an argument. We also call the *setUpGame* method before starting each new game in the GUI class.

Inside the *setUpGame* method, we assign *newSide* to *side*. Rather than randomly generating each tile label, which would complicate this example, we assign the labels to the tiles in descending order using nested *for* loops (lines 33–41). We set the empty cell to the last cell in the grid at lines 42–43.

The *tryToPlay* method (lines 71–89) first checks if the play is legal by calling the *possibleToPlay* method (line 77). If the *possibleToPlay* method returns *true*, we proceed with the play and return *true*; otherwise, we return *false*. Playing means swapping the values of the empty cell (*emptyRow*, *emptyCol*) and the cell that was just played, represented by the two parameters of *tryToPlay*, *row*, and *col*.

The *possibleToPlay* method is coded at lines 91–101. If the play is legal—that is, if the tile played is within one cell of the empty cell—the method returns *true*; otherwise, the method returns *false*.

The *won* method (lines 103–119) checks if the tiles are in order. If so, the *won* method returns *true*; otherwise, the method returns *false*.

As in the chessboard example, when the user clicks on a button, we need to identify the row and the column of that button. In order to avoid looping through all the buttons, we create a class that *extends* the *JButton* class and add two instance variables for the row and column indexes. In this way, when a button belonging to a two-dimensional array of buttons is clicked, we can access its row and column instance variables in order to retrieve its row and column indexes within the array.

Example 16.17 shows the *GridButton* class, which extends the *JButton* class and adds two instance variables, *row* and *column* (lines 9–10). The constructor at lines 12–22 allows the client to create a *GridButton* object at a specified row and column. Note that this class is not specific to the tile puzzle application and can be reused for other applications involving an array of buttons placed on a grid.

Example 16.17 The *GridButton* Class

```java
/** GridButton class
*   Anderson, Franceschi
*/

import javax.swing.JButton;

public class GridButton extends JButton
{
 private int row;
 private int column;

 /** Constructor
 * @param title
 * @param newRow row
 * @param newColumn column
 */
 public GridButton( String title, int newRow, int newColumn )
 {
  super( title );
  setRow( newRow );
  setColumn( newColumn );
 }

 /** getRow method, accessor for row
 * @return row
 */
 public int getRow( )
 {
   return row;
 }

 /** getColumn method, accessor for column
 * @return column
 */
 public int getColumn( )
 {
   return column;
 }

 /** setRow method, mutator for row
 * @param newRow, an int
 */
 public void setRow( int newRow )
 {
   if ( newRow >= 0 )
     row = newRow;
 }

 /** setColumn method
 * @param newColumn, an int
 */
 public void setColumn( int newColumn )
 {
   if ( newColumn >= 0 )
     column = newColumn;
 }
}
```

Description

Figure 16.21 shows the UML diagram for the *TilePuzzle* and *TilePuzzleViewController* classes.

Figure 16.21 UML Diagram for *TilePuzzle* and *TilePuzzleViewController*

Description

Example 16.18 shows the *TilePuzzleViewController* class; again, it combines the View and the Controller.

Example 16.18 The *TilePuzzleViewController* Class

```
/** Using GridLayout dynamically
*   Anderson, Franceschi
*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Random;

public class TilePuzzleViewController extends JFrame
{
 private Container contents;

 private GridButton [ ][ ] squares;
 private TilePuzzle game; // the tile puzzle game

 public TilePuzzleViewController( TilePuzzle newGame )
 {
  super( "The Tile Puzzle Game" );
  game = newGame;
  setUpGameGUI( );
 }

 public void setUpGameGUI( )
 {
  contents = getContentPane( );
  contents.removeAll( ); // remove all components
  contents.setLayout( new GridLayout( game.getSide( ),
                                      game.getSide( ) ) );
  squares = new GridButton [game.getSide( )][game.getSide( )];

  ButtonHandler bh = new ButtonHandler( );

  // for each button: instantiate button with appropriate button label,
  // add to container, and register listener
  for ( int i = 0; i < game.getSide( ); i++ )
  {
   for ( int j = 0; j < game.getSide( ); j++ )
   {
    squares[i][j] = new GridButton( game.getTiles( )[i][j], i, j );
    contents.add( squares[i][j] );
    squares[i][j].addActionListener( bh );
   }
  }

  setVisible( true );
 }

 public void update( )
 {
  for ( int i = 0; i < game.getSide( ); i++ )
  {
   for ( int j = 0; j < game.getSide( ); j++ )
   {
    squares[i][j].setText( game.getTiles( )[i][j] );
   }
  }

  if ( game.won( ) )
  {
   JOptionPane.showMessageDialog( TilePuzzleViewController.this,
         "Congratulations! You won!\nSetting up new game" );
   Random random = new Random( );
   int sideOfPuzzle = 3 + random.nextInt( 4 );
   game.setUpGame( sideOfPuzzle );
   setUpGameGUI( );
  }
 }

 private class ButtonHandler implements ActionListener
 {
  public void actionPerformed( ActionEvent ae )
  {
   GridButton button = ( GridButton ) ae.getSource( );
   if ( game.tryToPlay( button.getRow( ), button.getColumn( ) ) )
     update( );
  }
 }
}
```

Description

Each tile in the game is a *GridButton*. The instance variable *squares* (line 14) will hold a two-dimensional array of *GridButtons* so that each element of *squares* will be a cell in the game grid. The *TilePuzzle* instance variable *game*, declared at line 15, represents the Model, the tile puzzle game that we will display inside the window. When the user plays, we will call the various methods of the *TilePuzzle* class to enforce the rules of the game.

The constructor (lines 17–22) instantiates *game* and calls the *setUpGameGUI* method. The constructor and the *setUpGameGUI* method define the View. The *private* class *ButtonHandler* (lines 70–78) and the *update* method (lines 49–68) make up the Controller.

The *setUpGameGUI* method (lines 24–44) displays the game in its starting position. We first remove all the components from the content pane by calling the *removeAll* method of the *Container* class (line 27) and set the layout manager (lines 28–29) as a *GridLayout* with a grid size equal to the *side* instance variable of *game*. We instantiate the *squares* array and our event handler at lines 30 and 32. After that, we use nested *for* loops (lines 34–44) to instantiate each button, add it to the container, and register the event handler. The *squares* array parallels the *tiles* array of the *TilePuzzle* class; the *squares* buttons are labeled with the values of *tiles*.

When a button is clicked, our *ButtonHandler actionPerformed* method is executed. To determine which button was clicked, we call the *getSource* method using the *ActionEvent* event object that was sent to the method (line 74). We then need to determine whether the button clicked was a legal play. To do this, we call the *tryToPlay* method at line 75 with the row and column instance variable values of the button. If that method returns *true*, the play was legal and the model has changed (i.e., a tile has been moved). We then call the *update* method to make the view (i.e., the buttons) reflect the model (line 76).

The *update* method first updates the *squares* button array (lines 51–57). We then test if the current move solved the puzzle by calling the *won* method (line 59) of the *TilePuzzle* class. If the *won* method returns *true*, we congratulate the user by popping up a dialog box at lines 61–62. Note that we use *TilePuzzleViewController.this* as the parent component in the *showMessageDialog* method. This makes the dialog appear in the center of our application window.

We then randomly generate a grid size between three and six (lines 63–64) for the next game and call the *setUpGame* method (line 65) and the *setUpGameGUI* method (line 66) to begin a new game with the new grid size.

Finally, the *PlayTilePuzzle* class, shown in **Example 16.19**, includes the *main* method to create a *TilePuzzleViewController* application showing a *TilePuzzle* game. **Figure 16.22** shows a run of this game after the user has moved some tiles.

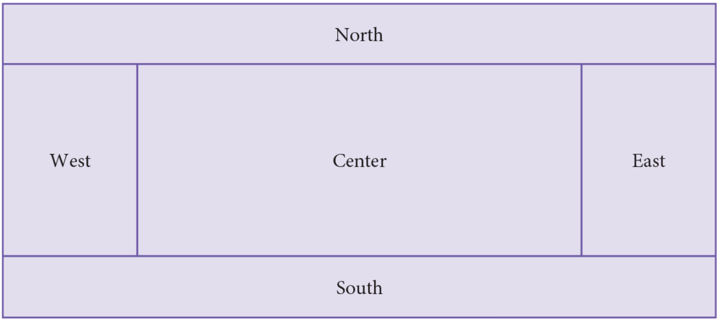
Example 16.19 The *PlayTilePuzzle* Class

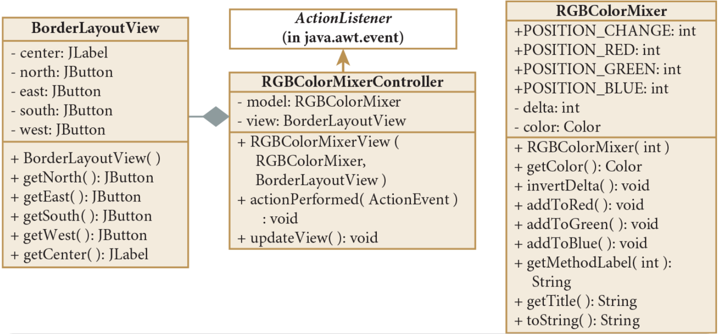
```java
/** PlayTilePuzzle class
*     Anderson, Franceschi
*/

import javax.swing.JFrame;

public class PlayTilePuzzle extends JFrame
{
 public static void main( String [ ] args )
 {
  TilePuzzle puzzle = new TilePuzzle( 3 );
  TilePuzzleViewController puzzleGui
    = new TilePuzzleViewController( puzzle );
  puzzleGui.setSize( 300, 300 );
  puzzleGui.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

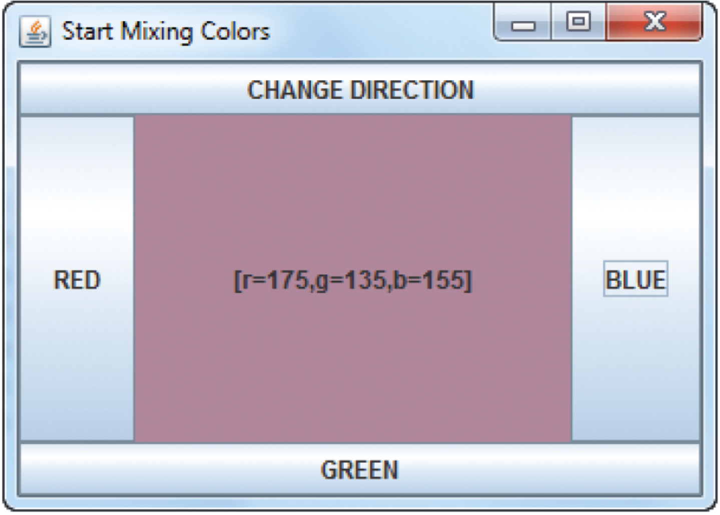
Description

Figure 16.22 Tile Puzzle Game in Progress

Description


North
West
Center
East
South


BorderLayoutView
- center: JLabel
- north: JButton
- east: JButton
- south: JButton
- west: JButton
+ BorderLayoutView( )
+ getNorth( ): JButton
+ getEast( ): JButton
+ getSouth( ): JButton
+ getWest( ): JButton
+ getCenter( ): JLabel
ActionListener
(in java.awt.event)
RGBColorMixerController
- model: RGBColorMixer
- view: BorderLayoutView
+ RGBColorMixerView (
RGBColorMixer,
BorderLayoutView)
+ actionPerformed(ActionEvent)
: void
+ updateView( ): void
RGBColorMixer
+POSITION_CHANGE: int
+POSITION_RED: int
+POSITION_GREEN: int
+POSITION_BLUE: int
- delta: int
- color: Color
+ RGBColorMixer(int)
+ getColor( ): Color
+ invertDelta( ): void
+ addToRed( ): void
+ addToGreen( ): void
+ addToBlue( ): void
+ getMethodLabel(int):
String
+ getTitle( ): String
+ toString( ): String


Start Mixing Colors
CHANGE DIRECTION
RED
BLUE
GREEN

[r=175,g=135,b=155]

16.13 Layout Managers: *BorderLayout*

A *BorderLayout* organizes a container into five areas: north, south, west, east, and center, with each area holding at most one component. Thus, for each area, you can add one component or no component. The size of each area expands or contracts depending on the size of the component in that area, the sizes of the components in the other areas, and whether the other areas contain a component. **Figure 16.23** shows the areas in a sample border layout.

Figure 16.23 *BorderLayout* Areas

Description

In contrast to the *FlowLayout* and *GridLayout* layout managers, the order in which we add components to a *BorderLayout* is not important. We use a second argument in the *add* method to specify the area in which to place the component. This *add* method of the *Container* class has the API shown in **Table 16.24**.

The *Container add* Method for Using a *BorderLayout*

Return Value	**Method Name and Argument List**
`void`	`add( Component c, Object borderlayoutArea )` adds the component *c* to the container. The area defined by *borderlayoutArea* can be specified using any of the following *static String* constants in the *BorderLayout* class: *NORTH, SOUTH, EAST, WEST, CENTER.*

Description

Two *BorderLayout* constructors are shown in **Table 16.25.** The *BorderLayout* is the default layout for a *JFrame* class, so if we want to use a border layout for our GUI application, we do not need to instantiate a new layout manager.

BorderLayout Constructors

Constructors

Class	**Constructor**
`BorderLayout`	`BorderLayout( )` creates a new border layout with no gaps between components.
`BorderLayout`	`BorderLayout( int hGap, int vGap )` creates a border layout with a horizontal gap of *hGap* pixels between components and a vertical gap of *vGap* pixels between components. Unlike *GridLayout*, horizontal gaps are not placed at the left and right edges, and vertical gaps are not placed at the top and bottom edges.

Description

Our next example uses a *BorderLayout* to illustrate how we can mix red, green, and blue colors to create a new color. We place buttons in the west, south, and east positions to represent the colors red, green, and blue. We place a label in the center position to show the color being made. Color values are between 0 and 255. When the user clicks on one of the buttons, we change the corresponding amount of that color by a specified increment. For example, if the amount of red is 100 and the incremental amount is 10, and the user clicks on the Red button, the value of red will become 110.

In the north position, we place a button "Change Direction," which will make the incremental amount toggle between a positive and a negative value; for example, if the incremental amount is 10 and the user clicks the button, the incremental amount becomes −10.

In this example, we use the MVC architecture and we separate the View from the Controller, thus making the View class reusable, in addition to the Model. **Figure 16.24** shows the UML diagram for *RGBColorMixer* (the Model), *BorderLayoutView* (the View), and *RGBColorMixerController* (the Controller).

Figure 16.24 The UML Diagram for Classes of This Application

Description

We first code our Model, the *RGBColorMixer* class (**Example 16.20**), a class that encapsulates the making of a color based on the RGB system by changing one color component at a time by a given increment. This class is a standalone class and can be reused in another application in a different context. The instance variable *color* (line 15), a *Color*, stores the color that we are making. The instance variable *delta* (line 14), an *int*, stores the amount by which a color component of *color* can be incremented or decremented at a time; *delta* can be either positive or negative. The value of a color component is between *0* and *255* included, so we only allow values between –*255* and *255* for *delta* (line 22). If *delta* is positive, the starting color is black (*0*, *0*, *0*), and if *delta* is negative, the starting color is white (*255*, *255*, *255*) (lines 26–29).

Example 16.20 The *RGBColorMixer* Class

```java
/** RGBColorMixer class
*   Anderson, Franceschi
*/

import java.awt.Color;

public class RGBColorMixer
{
 public static int POSITION_CHANGE = 0;
 public static int POSITION_RED = 1;
 public static int POSITION_GREEN = 2;
 public static int POSITION_BLUE = 3;

 private int delta; // color increment
 private Color color; // mixed color

 /** Constructor
 * @param newDelta delta
 */
 public RGBColorMixer( int newDelta )
 {
  if ( newDelta < -255 || newDelta > 255 || newDelta == 0 )
    delta = 1;
  else
    delta = newDelta;
  if ( delta > 0 )
    color = Color.BLACK;
  else
    color = Color.WHITE;
 }

 /** getColor method
 * @return color
 */
 public Color getColor( )
 {
  return color;
 }

 /** invertDelta method
 * sets delta to its opposite
 */
 public void invertDelta( )
 {
  delta = -delta;
 }

 /** addToRed method
 * add delta to red component of color
 */
 public void addToRed( )
 {
  if ( color.getRed( ) + delta > 255 )
    color = new Color( 255, color.getGreen( ), color.getBlue( ) );
  else if ( color.getRed( ) + delta < 0 )
    color = new Color( 0, color.getGreen( ), color.getBlue( ) );
  else
    color = new Color( color.getRed( ) + delta, color.getGreen( ),
                       color.getBlue( ) );
 }

 /** addToGreen method
 * add delta to green component of color
 */
 public void addToGreen( )
 {
  if ( color.getGreen( ) + delta > 255 )
    color = new Color( color.getRed( ), 255, color.getBlue( ) );
  else if ( color.getGreen( ) + delta < 0 )
    color = new Color( color.getRed( ), 0, color.getBlue( ) );
  else
    color = new Color( color.getRed( ), color.getGreen( ) + delta,
                       color.getBlue( ) );
 }

 /** addToBlue method
 * add delta to blue component of color
 */
 public void addToBlue( )
 {
  if ( color.getBlue( ) + delta > 255 )
    color = new Color( color.getRed( ), color.getGreen( ), 255 );
  else if ( color.getBlue( ) + delta < 0 )
    color = new Color( color.getRed( ), color.getGreen( ), 0 );
  else
    color = new Color( color.getRed( ), color.getGreen( ),
                       color.getBlue( ) + delta );
 }

  /** getMethodLabel method
 * @param position 0: Change, 1: Red, 2: Green, 3: Blue
 * @return a String to use as the text to label
 *         the calling of one of the methods
 *         invertDelta, addToRed, addToGreen, and addToBlue
 */
 public String getMethodLabel( int position )
 {
  if ( position == POSITION_CHANGE )
    return "CHANGE DIRECTION";
  else if ( position == POSITION_RED )
    return "RED";
  else if ( position == POSITION_GREEN )
    return "GREEN";
  else if ( position == POSITION_BLUE )
    return "BLUE";
  else
    return "";
 }

 /** getTitle method
  * @return a String to use as title for this object
 */
 public String getTitle( )
 {
   return "Start Mixing Colors";
 }

 /** toString method
 * @return a String representation of color
 */
 public String toString( )
 {
  String colorText = color.toString( );
  int bracketIndex = colorText.indexOf( "[" );
  if ( bracketIndex >= 0 )
    colorText = colorText.substring( bracketIndex, colorText.length( ) );
  return colorText;
 }
}
```

Description

The *invertDelta* method (lines 40–46) changes the value of *delta* to its opposite. This will be useful, for example, if delta is positive and any of the color components reaches the maximum value of 255, or if delta is negative and any of the color components reaches the minimum value of 0.

The *addToRed* method (lines 48–60) adds *delta* to the red component of *color*, keeping that value between *0* and *255*. The *addToGreen* and *addToBlue* methods are similar and update the green and blue components of *color*, respectively. These methods call the *getRed*, *getGreen*, and *getBlue* accessor methods of the *Color* class to get the current *int* value of each color component.

The *getMethodLabel* method (lines 90–108) returns a *String* that can be used as a label in a graphical user interface for the methods *invertDelta*, *addToRed*, *addToGreen*, and *addToBlue*. Although we know that we will be using a *BorderLayout* for this application, we design *RGBColorMixer* so that it can be reused with other Views and layouts. Thus, we do not use the *BorderLayout String* constants *NORTH, WEST, SOUTH*, and *EAST* as possible arguments passed to this method. Instead, we provide static constants (lines 9–12) to represent possible changes in *delta* (*POSITION_CHANGE*), the red component of this color (*POSITION_RED*), the green component of this color (*POSITION_GREEN*), and the blue component of this color (*POSITION_BLUE*). These constants can be used when calling the *getMethodLabel* method.

The *getTitle* method (lines 110–116) returns a *String* that can be used as the title of an application using this Model. The *toString* method (lines 118–128) returns a *String* representation of *color*.

Example 16.21 shows the *BorderLayoutView* class, which represents the View for this application. Note that the Controller is not included in this class and this class is generic; it can be used for many other applications.

Example 16.21 The *BorderLayoutView* Class

```
/** BorderLayoutView class
*   Reusable generic frame using a BorderLayout
*   Four buttons occupy the North, East, South, and West positions
*   A label with centered text occupies the center position
*   Accessors are provided so that a Controller can access
*   the four buttons and the label
*   Anderson, Franceschi
*/

import javax.swing.*;
import java.awt.*;

public class BorderLayoutView extends JFrame
{
 private JButton north, east, south, west;
 private JLabel center;

 public BorderLayoutView( )
 {
  super( );
  Container contents = getContentPane( );
  contents.setLayout( new BorderLayout( ) ); // optional

  // instantiate button objects
  north = new JButton( );
  east = new JButton( );
  south = new JButton( );
  west = new JButton( );

  center = new JLabel( "", SwingConstants.CENTER );
  center.setOpaque( true );

  // order of adding components is not important
  contents.add( north, BorderLayout.NORTH );
  contents.add( east, BorderLayout.EAST );
  contents.add( south, BorderLayout.SOUTH );
  contents.add( west, BorderLayout.WEST );
  contents.add( center, BorderLayout.CENTER );

  setVisible( true );
 }

 public JButton getNorth( )
 {
  return north;
 }

 public JButton getEast( )
 {
  return east;
 }

 public JButton getSouth( )
 {
  return south;
 }

 public JButton getWest( )
 {
  return west;
 }

 public JLabel getCenter( )
 {
  return center;
 }
}
```

Description

We declare four buttons at line 15; we will place one button each in the north, east, south, and west areas of the border layout. We use the label *center*, declared at line 16, to display something in the center area.

At line 22, we set the layout of the content pane to be a *BorderLayout*. Note that this statement is optional, because *BorderLayout* is the default layout manager for a subclass of the *JFrame* class.

When instantiating *center* at lines 30–31, we use an overloaded *JLabel* constructor. We use the *static int* constant *CENTER* of the *SwingConstants* interface to center the text of the label within the component. By default, the text for a *JLabel* is left-aligned. Because the text on a *JButton* is centered by default, we can simply use the constructor with one argument when we instantiate the *JButtons*.

We add the four buttons and the label to the content pane in the appropriate areas at lines 33–38. The order in which we add the components is not important because we specify a border layout area as the second argument of the *add* method.

We also provide accessor methods for the four buttons and the label so that they can be updated from outside the class.

Note that the Model and the View are completely independent and can be reused separately in other applications.

Example 16.22 shows the *RGBColorMixerController* class, which is the Controller for this application. At lines 10–11, we declare *model*, an *RGBColorMixer* instance variable, and *view*, a *BorderLayoutView*, our two instance variables. They enable us to get user input from the View, call the various methods of the Model (the *RGBColorMixer* class), and update the View.

Example 16.22 The *RGBColorMixerController* Class

```
/** RGBColorMixerController class
*   Anderson, Franceschi
*/

import java.awt.event.*;
import javax.swing.*;

public class RGBColorMixerController implements ActionListener
{
 private RGBColorMixer model;
 private BorderLayoutView view;

 public RGBColorMixerController( RGBColorMixer newModel,
        BorderLayoutView newView )
 {
  model = newModel;
  view = newView;

  view.setTitle( model.getTitle( ) );

  JButton north = view.getNorth( ), west = view.getWest( ),
          south = view.getSouth( ), east = view.getEast( );

  north.setText( model.getMethodLabel( RGBColorMixer.POSITION_CHANGE ) );
  west.setText( model.getMethodLabel( RGBColorMixer.POSITION_RED ) );
  south.setText( model.getMethodLabel( RGBColorMixer.POSITION_GREEN ) );
  east.setText( model.getMethodLabel( RGBColorMixer.POSITION_BLUE ) );

  north.addActionListener( this );
  west.addActionListener( this );
  south.addActionListener( this );
  east.addActionListener( this );

  updateView( );
 }

 public void actionPerformed( ActionEvent ae )
 {
  // update the model
  if ( ae.getSource( ) == view.getNorth( ) )
    model.invertDelta( );
  else if ( ae.getSource( ) == view.getWest( ) )
    model.addToRed( );
  else if ( ae.getSource( ) == view.getSouth( ) )
    model.addToGreen( );
  else if ( ae.getSource( ) == view.getEast( ) )
    model.addToBlue( );
  // update the view
  updateView( );
 }

 /** updateView method
 * updates the color and the text of the center label
 */
 public void updateView( )
 {
  JLabel label = view.getCenter( );
  label.setBackground( model.getColor( ) );
  label.setText( model.toString( ) );
 }
}
```

Description

The constructor (lines 13–35) accepts two parameters that it assigns to *model* and *view*. It sets the title of the frame at line 19, retrieving it from the Model. We use the north button to invert the value of *delta* in *model*, and the west, south, and east buttons to change the red, green, and blue components of *color* in *model*, respectively. Thus, we label the four buttons of the View at lines 24–27 by calling the *getMethodLabel* method with *model*, passing the appropriate position argument. We set up event handling at lines 29–32, registering *this* object (*RGBColorMixerController implements ActionListener*) on the four buttons of the View.

In our *actionPerformed* method (lines 37–50), we need to update the Model and the View. To update *model*, we call the appropriate method of *RGBColorMixer*, depending on which button was clicked. For example, if the west button was clicked, we call *addToRed* (line 43). At lines 48–49, we update the View by calling *updateView*.

In our *updateView* method (lines 52–60), we update the label located in the center of the View, changing its color to the color of *model*, and changing its text to the *String* representation of the Model.

Finally, the *MixColors* class, shown in **Example 16.23**, includes the *main* method to create a color mixing application. Note that we set the size of the View inside *main*, so that we do not have hard-coded dimensions inside the View and Controller classes. **Figure 16.25** shows a run of this example, after the user has clicked on the various buttons a few times.

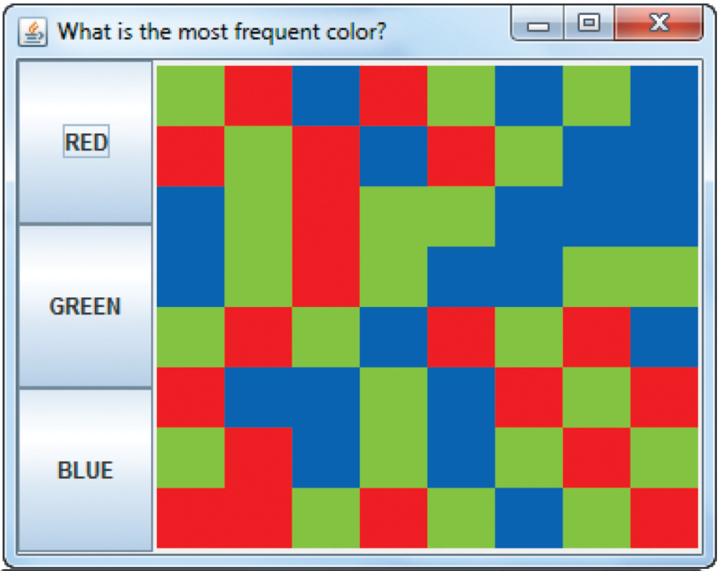
Example 16.23 The *MixColors* Class

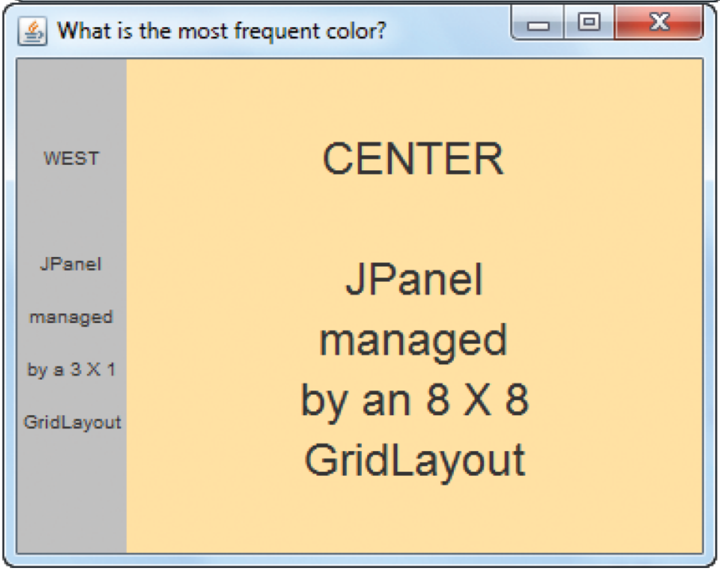
```
/** MixColors class
*   Anderson, Franceschi
*/

import javax.swing.JFrame;

public class MixColors
{
 public static void main( String [ ] args )
 {
  // Instantiate the Model
  RGBColorMixer colorMixer = new RGBColorMixer( -10 );
  // Instantiate the View
  BorderLayoutView colorMixerGui = new BorderLayoutView( );
  colorMixerGui.setSize( 350, 250 );
  colorMixerGui.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
  // Instantiate the Controller
  RGBColorMixerController controller
    = new RGBColorMixerController( colorMixer, colorMixerGui );
 }
}
```

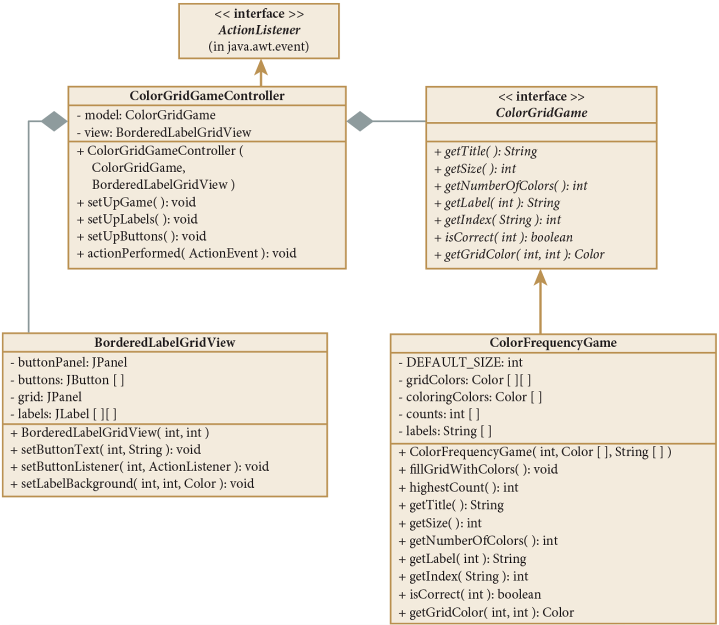
Description

Figure 16.25 Running the *MixColors* Application

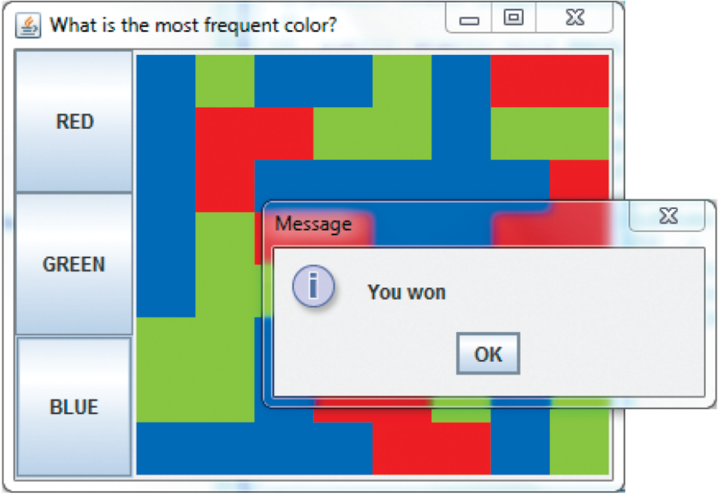
Description


What is the most frequent color?
RED
GREEN
BLUE


What is the most frequent color?
WEST
JPanel
managed
by a 3 X 1
GridLayout
CENTER
JPanel
managed
by an 8 X 8
GridLayout



<< interface >>
ActionListener
(in java.awt.event)
ColorGridGameController
- model: ColorGridGame
- view: BorderedLabelGridView
+ ColorGridGameController (ColorGridGame, BorderedLabelGridView)
+ setUpGame( ): void
+ setUpLabels( ): void
+ setUpButtons( ): void
+ actionPerformed(ActionEvent): void
<< interface >>
ColorGridGame
+ getTitle( ): String
+ getSize( ): int
+ getNumberOfColors( ): int
+ getLabel(int): String
+ getIndex(String): int
+ isCorrect(int): boolean
+ getGridColor(int, int): Color
BorderedLabelGridView
- buttonPanel: JPanel
- buttons: JButton [ ]
- grid: JPanel
- labels: JLabel [ ][ ]
+ BorderedLabelGridView(int, int)
+ setButtonText(int, String): void
+ setButtonListener(int, ActionListener): void
+ setLabelBackground(int, int, Color): void
ColorFrequencyGame
- DEFAULT_SIZE: int
- gridColors: Color [ ][ ]
- coloringColors: Color [ ]
- counts: int [ ]
- labels: String [ ]
+ ColorFrequencyGame(int, Color [ ], String [ ])
+ fillGridWithColors( ): void
+ highestCount( ): int
+ getTitle( ): String
+ getSize( ): int
+ getNumberOfColors( ): int
+ getLabel(int): String
+ getIndex(String): int
+ isCorrect(int): boolean
+ getGridColor(int, int): Color


What is the most frequent color?

Message

You won

RED
GREEN
BLUE
OK

16.14 Using Panels to Nest Components

Components can be nested. Indeed, since the *JComponent* class is a subclass of the *Container* class, a *JComponent* object is a *Container* object as well. As such, it can contain other components. We can use this feature to achieve more precise layouts.

The *JPanel* class is a general-purpose container, or a **panel**, and is typically used to hold other components. When nesting components, we usually place several components into a panel, and place the panel into the content pane of the current *JFrame*. Each panel has its own layout manager, which can vary from panel to panel, and the content pane for the *JFrame* application has its own layout manager, which can be different from any of the panels. We can even have multiple levels of nesting, as needed.

In this example, using three colors (red, green, and blue), we randomly generate an 8-by-8 grid of colors. The user tries to guess the most frequently occurring color by clicking on one of the buttons.

Figure 16.26a shows our window when the application begins, and **Figure 16.26b** shows the underlying layout of the window. We use a *BorderLayout* for the content panel and two

JPanels, each of which has a *GridLayout*. In the west position, we have a *JPanel* containing three buttons in a *GridLayout* having three rows and one column.

Figure 16.26a Running the *MixColors* Application

Description

Figure 16.26b The Underlying Nested Layout

Description

In the center position, we have defined another *JPanel* containing 64 labels managed by a *GridLayout* having eight rows and eight columns. We have not put any components in the north, south, or east areas of the *BorderLayout;* thus, the two areas we are using, west and center, adjust to fill the space.

Again, we use the MVC architecture and design our classes for reusability. This time, we also code the Controller so that it is reusable under certain conditions. Inside the *Controller* class, instead of using a class for the Model, we use an interface. Thus, the Controller is now reusable, along with the same View, with any Model that implements that interface. Note that we could also make the View an interface, but in order to keep this example relatively simple, we do not. To achieve reusability for the Controller, we use the following pattern for the *Controller* class:

```
public class Controller
{
 private ModelInterface model; // ModelInterface is an interface
 private View view;
 …
```

Description

The *ModelInterface* specifies several methods to be implemented that the Controller will call to manage the game. The *Model* class uses this pattern:

```
public class Model implements ModelInterface
{
 // Implements ModelInterface methods
 …
```

Description

We want the *Controller* class to be reusable with any class whose functionality is based on the user clicking buttons on the left panel with a grid of labels in the center position. We could use this class in an application where the user tries to guess the most frequent color, or the least frequent color, or with a different color system (such as HSL – Hue, Saturation, and Lightness), or the most frequent letter or number. Figure 16.27 shows the UML for the interface and classes involved in this example.

Figure 16.27 The UML diagram for the Various Classes and Interface of This Application

Description

Thus, we design an interface to support this type of application. **Example 16.24** shows the *ColorGridGame* interface. It specifies methods to retrieve the title for the grid, its size, its number of colors, the label of a color, the color index of a label (it is implied that the colors used in the grid of colors will come from an array of colors, and that there will be an array of *Strings* (labels) that parallels that array of colors), to test if an answer is the correct answer, and to access each color in the grid. We call these methods from the Controller class with the Model instance variable in order to manage the game.

Example 16.24 The *ColorGridGame* Interface

```
/* ColorGridGame interface
   Anderson, Franceschi
 */

import java.awt.Color;

public interface ColorGridGame
{
 public String getTitle( );
 public int getSize( );
 public int getNumberOfColors( );
 public String getLabel( int index );
 public int getIndex( String label );
 public boolean isCorrect( int index );
 public Color getGridColor( int row, int col );
}
```

Description

We now code our Model for the color frequency game. In this application, we generate colors for our grid of labels.

Example 16.25 shows the *ColorFrequencyGame* class, which *implements ColorGridGame* (line 8). Its *gridColors* instance variable (line 11) stores a two-dimensional array, or grid, of colors. The *coloringColors* and *counts* arrays (lines 12–13) are parallel arrays: *coloringColors* stores the colors used for coloring the grid, and *counts* stores the frequency of each color. The instance variable *labels* (line 15) is an array of *Strings* that also parallels the *coloringColors* array; it is meant to be used as labels for the buttons.

Example 16.25 The *ColorFrequencyGame* Class

```java
/** ColorFrequencyGame class
   Anderson, Franceschi
*/

import java.util.Random;
import java.awt.Color;

public class ColorFrequencyGame implements ColorGridGame
{
 private final int DEFAULT_SIZE = 8;
 private Color [ ][ ] gridColors;
 private Color [ ] coloringColors;
 private int [ ] counts; // color frequencies in gridColors

 private String [ ] labels;

 /** Constructor
 * @param size number of rows and columns in gridColors
 * @param colors coloringColors
 * @param newLabels labels
 */
 public ColorFrequencyGame( int size, Color [ ] colors,
                            String [ ] newLabels )
 {
  if ( size <= 0 )
    size = DEFAULT_SIZE;
  gridColors = new Color[size][size];
  coloringColors = colors;
  counts = new int[coloringColors.length];
  fillGridWithColors( );
  labels = newLabels;
 }

 /** fillGridWithColors method
 * randomly fills gridColors with colors from coloringColors
 */
 public void fillGridWithColors( )
 {
  Random random = new Random( );
  for ( int i = 0; i < gridColors.length; i++ )
  {
   for ( int j = 0; j < gridColors[i].length; j++ )
   {
    int colorIndex = random.nextInt( coloringColors.length );
    gridColors[i][j] = coloringColors[colorIndex];
    counts[colorIndex] += 1;
   }
  }
 }

 /** getSize method
 * @return length of gridColors
 */
 public int getSize( )
 {
  return gridColors.length;
 }

 /** getNumberOfColors method
 * @return length of coloringColors
 */
 public int getNumberOfColors( )
 {
  return coloringColors.length;
 }

 /** getGridColor method
 * @param row, an int, the row index
 * @param col, an int, the column index
 * @return gridColor[row][col]
 */
 public Color getGridColor( int row, int col )
 {
  return gridColors[row][col];
 }

 /** highestCount method
 * @return the highest color frequency in the grid
 */
 public int highestCount( )
 {
  int max = counts[0];
  for ( int i = 1; i < counts.length; i++ )
  {
   if ( counts[i] > max )
     max = counts[i];
  }
  return max;
 }

 /** isCorrect method
 * @param index, an int
 * @return true if the frequency of the color for index
 *         is the highest, false otherwise
 */
 public boolean isCorrect( int index )
 {
  return counts[index] == highestCount( );
 }

 /** getTitle method
 * @return a String representing a title for this object
 */
 public String getTitle( )
 {
  return "What is the most frequent color?";
 }

 /** getLabel method
 * @param index, an int
 * @return labels[index]
 */
 public String getLabel( int index )
 {
  return labels[index];
 }

 /** getIndex method
 * @param label, a String
 * @return index of label in labels
 */
 public int getIndex( String label )
 {
  for ( int i = 0; i < labels.length; i++ )
  {
   if ( labels[i].equals( label ) )
     return i;
  }
  return -1;
 }
}
```

Description

The constructor (lines 17–32) instantiates all three arrays and calls the *fillGridWithColors* method at line 30. The *fillGridWithColors* method (lines 34–49) fills the grid of colors with colors randomly generated from the *coloringColors* array and keeps track of their respective frequency in the *counts* array (line 46).

To determine whether the user has chosen the correct color, we provide the *isCorrect* method (lines 91–99) that takes as a parameter the index of the selected color. It returns *true* if the value in *counts* at that index is maximal, *false* otherwise. The *highestCount* method (lines 77–89) calculates and returns the maximum value in the *counts* array.

The *getLabel* method (lines 109–116) returns the element of the array *labels* at an index, the parameter of the method; that is, the label to be used with a color located at that index within the array *coloringColors*. The *getIndex* method (lines 118–130) is the opposite method: It returns the index within the array *labels* of a *String*, the parameter of the method; the color corresponding to that label is the element in the array *coloringColors* at that index.

Example 16.26 shows the *BorderedLabelGridView* class. It is managed by a *BorderLayout*, which contains two *JPanels*, one in the west and one in the center position. The other positions within the *BorderLayout* are not used. The west panel contains an array of buttons, arranged vertically, managed by a *GridLayout*, and the center panel contains a grid of labels, managed by another *GridLayout*. Thus, our instance variables are two panels, an array of buttons, and a two-dimensional array of labels (lines 17–20).

Example 16.26 The *BorderedLabelGridView* Class

```
/** BorderedLabelGridView class
*   Reusable generic frame using a BorderLayout
*   containing the west and center panels
*   The west panel is made up of a vertical array of buttons
*   The center panel is made up of a grid of labels
*   Accessors are provided so that a Controller can access
*   the array of buttons and the 2-dim array (grid) of labels
*   Anderson, Franceschi
*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionListener;

public class BorderedLabelGridView extends JFrame
{
 private JPanel buttonPanel; // left, holds array of buttons
 private JButton[ ] buttons;
 private JPanel grid; // center, holds grid of labels
 private JLabel[ ][ ] labels; // grid of labels

 public BorderedLabelGridView( int numberOfButtons, int gridSize )
 {
  super( );

  Container contents = getContentPane( );
  contents.setLayout( new BorderLayout( ) );

  grid = new JPanel( );
  GridLayout gameGrid = new GridLayout( gridSize, gridSize );
  grid.setLayout( gameGrid );

  labels = new JLabel[gridSize][gridSize];
  for ( int i = 0; i < labels.length; i++ )
  {
    for ( int j = 0; j < labels[i].length; j++ )
    {
     labels[i][j] = new JLabel( );
     labels[i][j].setOpaque( true );
     grid.add( labels[i][j] );
    }
  }

  buttonPanel = new JPanel( );
  GridLayout buttonGrid = new GridLayout( numberOfButtons, 1 );
  buttonPanel.setLayout( buttonGrid );

  buttons = new JButton[numberOfButtons];
  for ( int i = 0; i < buttons.length; i++ )
  {
    buttons[i] = new JButton( );
    buttonPanel.add( buttons[i] );
  }

  contents.add( BorderLayout.WEST, buttonPanel );
  contents.add( BorderLayout.CENTER, grid );
  setVisible( true );
 }

 public void setButtonText( int row, String text )
 {
  buttons[row].setText( text );
 }

 public void setButtonListener( int row, ActionListener al )
 {
  buttons[row].addActionListener( al );
 }

 public void setLabelBackground( int row, int col, Color color )
 {
  labels[row][col].setBackground( color );
 }
}
```

Description

We first define the layout manager for our content panel to be a *BorderLayout* at line 27. Then we set up the panel for the center position at lines 29–31, by instantiating the *JPanel* instance variable named *grid* and setting its layout manager as a *GridLayout* using the constructor parameter *gridSize* for the number of rows and columns. At lines 33–42, we instantiate the labels and add them to the grid.

At lines 44–46, we set up the panel for the west position by instantiating the *JPanel* instance variable named *buttonPanel* and setting its layout manager as a *GridLayout* with *numberOfButtons* rows and one column. At lines 48–53, we instantiate the buttons and add them to *buttonPanel*.

We add *buttonPanel* and *grid* to the content pane at lines 55–56 in the west and center positions, respectively. Finally, we provide accessor methods for the arrays of buttons and labels so that they can be accessed from outside the class.

We also provide methods to set the text and set an *ActionListener* for each button in the array *buttons*, and set the background color of each label in the array *labels*. By calling these methods, the Controller can change the View (based on data from the Model).

Example 16.27 shows the *ColorGridGameController* interface. At lines 11–12, we declare *model*, a *ColorGridGame*, and *view*, a *BorderedLabelGridView*, our two instance variables. They enable us to get user input from the View, call the appropriate methods of the Model, and update the View accordingly.

Example 16.27 The *ColorGridGameController* Class

```java
/** ColorGridGameController class
*   Anderson, Franceschi
*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ColorGridGameController implements ActionListener
{
 private ColorGridGame model;
 private BorderedLabelGridView view;

 public ColorGridGameController( ColorGridGame newModel,
                                 BorderedLabelGridView newView )
 {
  model = newModel;
  view = newView;
  setUpGame( );
 }

 public void setUpGame( )
 {
  view.setTitle( model.getTitle( ) );
  setUpLabels( );
  setUpButtons( );
 }

 public void setUpLabels( )
 {
  for ( int i = 0; i < model.getSize( ); i++ )
    for ( int j = 0; j < model.getSize( ); j++ )
      view.setLabelBackground( i, j, model.getGridColor( i, j ) );
 }

 public void setUpButtons( )
 {
  for ( int i = 0; i < model.getNumberOfColors( ); i++ )
  {
    view.setButtonText( i, model.getLabel( i ) );
    view.setButtonListener( i, this );
  }
 }

 public void actionPerformed( ActionEvent ae )
 {
  JButton button = ( JButton ) ae.getSource( );
  int index = model.getIndex( button.getText( ) );

  if ( model.isCorrect( index ) )
    JOptionPane.showMessageDialog( view, "You won" );
  else
    JOptionPane.showMessageDialog( view, "Sorry, you lost" );
  // disable buttons here
 }
}
```

Description

The constructor (lines 14–20) accepts two parameters that it assigns to *model* and *view*, and calls the *setUpGame* method at line 19. The *setUpGame* method (lines 22–27) sets the title of the View, then calls the *setUpLabels* and *setUpButtons* methods.

The *setUpLabels* method (lines 29–34) colors the grid of labels in the View using a double loop. At line 33, we set the background color of the current label by calling the *setLabelBackground* method with *view*, passing the row index, column index, and the background color for that label that we retrieve from the Model by calling the *getGridColor* method.

The *setUpButtons* method (lines 36–43) places a label on each button in the View and sets up event handling using a single loop. At line 40, we set the text for the current button, passing its index and the label that we retrieve from the Model by calling the *getLabel* method. At line 41, we register *this* object (i.e., the controller that implements *ActionListener*) on the current button.

We handle the events for the buttons in the *actionPerformed* method at lines 45–55. We first retrieve the button that originated the event by calling *getSource* at line 47. The *getSource* method returns an *Object*, so we need to type cast the return value to a *JButton*. We then retrieve the index of that button's corresponding label by calling the *getIndexOfLabel* method with *model*, passing the text of the button. We then check if the user won or lost by calling the *isCorrect* method of the *ColorGridGame* interface with *model*, passing that index. Depending on the result, we pop up a dialog box telling the user if he or she has won or lost (lines 50–53). Disabling the buttons after this is left as an exercise.

Finally, the *PlayColorCount* class, shown in **Example 16.28**, includes the *main* method to create a color count application. We declare an array of three *Colors* at line 12, declare a parallel array of *Strings* at line 13, create the *ColorFrequencyGame* object *game* at lines 14–15 with them, and create a *BorderedLabelGridView* object *gameGui* at lines 16–17. Using *game* and *gameGui*, we create a *ColorGridGameController* at lines 20–21. Note that *game* “is a” *ColorGridGame* object because *ColorFrequencyGame* inherits from *ColorGridGame*. **Figure 16.28** shows a run of this example, after the user has clicked on the button “BLUE.” To play the game with more colors, we could simply add colors and labels to the *colors* and *labels* array.

Example 16.28 The *PlayColorCount* Class

```java
/** PlayColorCount class
*   Anderson, Franceschi
*/

import javax.swing.JFrame;
import java.awt.Color;

public class PlayColorCount
{
 public static void main( String [ ] args )
 {
  Color [ ] colors = { Color.RED, Color.GREEN, Color.BLUE };
  String [ ] labels = { "RED", "GREEN", "BLUE" };
  ColorFrequencyGame game
    = new ColorFrequencyGame( 8, colors, labels );
  BorderedLabelGridView gameGui
    = new BorderedLabelGridView( colors.length, game.getSize( ) );
  gameGui.setSize( 380, 300 );
  gameGui.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
  ColorGridGameController controller
    = new ColorGridGameController( game, gameGui );
 }
}
```

Description

Figure 16.28 The User Clicked on the Button BLUE

Description

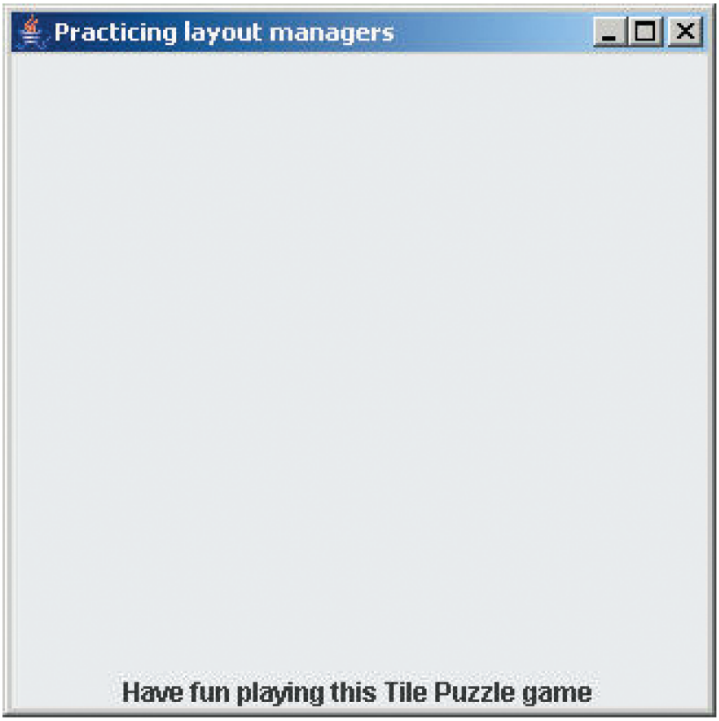
with these end-of-chapter questions

16.17.1	Multiple Choice Exercises Questions 9, 10, 11, 12, 13, 14, 15
16.17.2	Reading and Understanding Code Question 17
16.17.3	Fill In the Code Questions 30, 33, 34, 35, 36, 37, 38, 39, 40, 41
16.17.5	Debugging Area Question 48

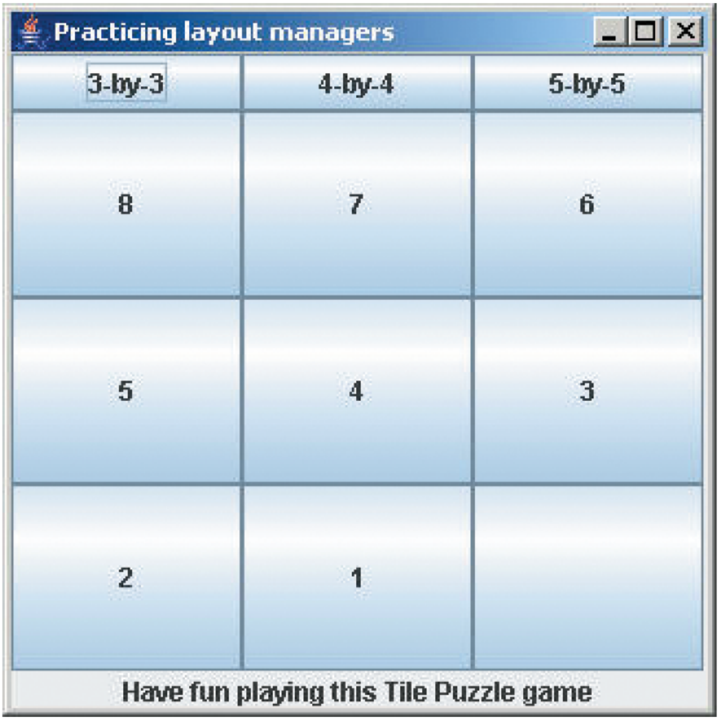
16.17.6	Write a Short Program Questions 58, 59, 60, 61, 62
16.17.8	Technical Writing Question 73

CODE IN ACTION

On the companion website you will find a Flash movie with a step-by-step illustration of working with layout managers. Click on the link to start the movie.


Practicing layout managers

Have fun playing this Tile Puzzle game


Practicing layout managers
3-by-3
4-by-4
5-by-5
8
7
6
5
4
3
2
1
Have fun playing this Tile Puzzle game

16.15 Programming Activity 2: Working with Layout Managers

In this Programming Activity, you will complete the implementation of a version of the Tile Puzzle game using a more complex GUI. As it stands, the application compiles and runs, but is missing a lot of code. **Figure 16.29** shows the window that will open when you run the application without adding your code.

Figure 16.29 The Starting Window When Running the Prewritten Code

Description

Once you have completed the five tasks of this Programming Activity, you should see the window in **Figure 16.30** when you run your program and click on the “3-by-3” button.

Figure 16.30 The Starting Window When the Activity Is Completed

Description

When you click on one of the buttons labeled “3-by-3”, “4-by-4”, or “5-by-5”, the tile puzzle will reset to a grid of that size.

In addition to the *TilePuzzle* class, we provide you with a prewritten *Game* class, which encapsulates a Tile Puzzle game in a panel. We have implemented the *Game* class as a *JPanel* component, so you can add it to your window as you would any other panel. It has two important methods, shown in **Table 16.26**. Thus, your job in this Programming Activity is not to write the game code, but to organize components in a window.

The *Game* Class API

Game **Class API**	
Constructor	
Class	**Constructor**
`Game`	`Game( int nSides )`
instantiates a tile puzzle as an *nSides*-by-*nSides* grid	
Method	
Return value	**Method name and argument list**
`void`	`setUpGame( int nSides )`
resets the grid as an *nSides*-by-*nSides* grid	

Description

Your job is to:

1. Declare a *JPanel top* and three *JButtons* that will be added to the “north” part of the window.
2. Set the layout managers for the *game* panel and the top panel.

3. Add the *top* and the *game* panels.
4. Code an appropriate *private* listener class.
5. Instantiate the listener and register it on the appropriate components.

Instructions

Copy the source files in the Programming Activity 2 folder for this chapter to a folder on your computer.

1. Write the code to declare the needed instance variables. Load the *NestedLayoutPractice.java* source file and search for five asterisks in a row (*****). This will position you to the instance variables declaration.

```
// ***** Task 1: declare a JPanel named top
// also declare three JButton instance variables
// that will be added to the JPanel top
// these buttons will determine the grid size of the game:
//  3-by-3, 4-by-4, or 5-by-5

// task 1 ends here
```

Description

1. Next, write the code to set the layout manager of the window and add the component *game* in the center of the window. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you inside the constructor.

```
 // ***** Task 2: student code starts here
 // instantiate the BorderLayout manager bl
// set the layout manager of the content pane contents to bl

game = new Game( 3 );  // instantiating the Game object
// add game to the center of the content pane

// task 2 ends here
```

Description

1. Next, write the code to instantiate the *JPanel* top component, set its layout, instantiate the buttons from task 1, add them to *top*, and finally add *top* as the north component for our overall window. In the *NestedLayoutPractice .java* source file, search again for five asterisks in a row (*****). This will position you inside the constructor.

```
// ***** Task 3: Student code restarts here

// instantiate the JPanel component named top

// set the layout of top to a 1-by-3 grid

// instantiate the JButtons that determine the grid size

// add the buttons to JPanel top

// add JPanel top to the content pane as its north component

 // task 3 ends here
```

Description

1. Next, write the code for the *private* inner class that implements the appropriate listener. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you between the constructor and the *main* method.

```
// ***** Task 4: Student code restarts here
// create a private inner class that implements ActionListener
// your method should identify which of the 3 buttons was the
//   source of the event.
// depending on which button was pressed,
//   call the setUpGame method of the Game class
//   with arguments 3, 4, or 5
// the API of that method is:
//   public void setUpGame( int nSides )
// task 4 ends here
```

Description

1. Next, write the code to declare and instantiate a listener, and register it on the appropriate components. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you inside the constructor.

```
// ***** Task 5: Student code restarts here
// Note: search for and complete Task 4 before performing this task
// declare and instantiate an ActionListener
// register the listener on the 3 buttons
//   that you declared in Task 1

 // task 5 ends here
```

Description

After completing each task, compile your code.

When you have finished writing all the code, compile the source code and run the *NestedLayoutPractice* application. Try clicking on the three buttons that you added.

DISCUSSION QUESTIONS

1. Identify the various layout managers you used and the panels they manage.
2. Explain why the east and west areas do not show up on the window.

© Lois GoBe/Shutterstock

CHAPTER REVIEW

16.16 Chapter Summary

- A graphical user interface allows the user to interact with an application through mouse clicks, mouse movements, the keyboard, and visual input components.
- The *JFrame* class provides the capabilities for creating a window that will hold GUI components.
- A constructor in a GUI application should call the constructor of the *JFrame* superclass, get an object reference to the content pane, set the layout manager, instantiate each component and add it to the window, set the size of the window, and make the window visible.
- A component is an object that has a graphical representation and that displays information, collects data from the user, or allows the user to initiate program functions.
- Java provides a set of GUI components in the *javax.swing* package.
- A *JLabel* component can display text or an image.
- The *FlowLayout* layout manager arranges components in rows from left to right, starting a new row whenever a newly added component does not fit into the current row. If the window is resized, the components are rearranged.
- Event-driven programming consists of setting up interactive

components and event handlers and responding to events generated by user interaction with the components.

- Use the **Model-View-Controller** architecture to make applications as reusable as possible. The **Model** encapsulates the application's functionality; the **View** encapsulates the user interface of the application; and the **Controller** is the middleman between the View and the Model.
- To allow a user to interact with a component, we need to instantiate an object of that class, write an event handler class (called a listener), instantiate an object of the event handler class, and register that listener on the component.
- Event handlers implement a listener interface in the *java.awt.event* or the *javax.swing.event* package. The listener methods receive as a parameter an event object, which encapsulates information about the user interaction. The *getSource* method can be called to determine which component fired the event.
- Event handlers usually are instantiated as *private* inner classes so the handler will have access to the components of the application.
- A *JTextField* component displays a text field for user input. A *JPasswordField* component accepts user input without echoing the characters typed. A *JTextArea* component displays a multiple-line text input area. Events fired by the *JTextField* and *JPasswordField* components generate an *ActionEvent* object and require an *ActionListener* to handle the event.
- A *JButton* component implements a command button used for initiating operations. Clicking on a *JButton* component generates an *ActionEvent* object and requires an *ActionListener* to handle the event.
- *JRadioButton* components allow the user to select one of several mutually exclusive options. Clicking on any radio button deselects any previously selected option. *JRadioButton* components need to be added to a *ButtonGroup*, which manages the mutual exclusivity of the buttons. *JCheckBox* components are toggle buttons;

successive clicks on a *JCheckBox* component alternately select and deselect that option. Events fired by these components generate an *ItemEvent* object and use an *ItemListener* to handle the event.

- A *JComboBox* implements a drop-down list. The user can select only one item from the list. A *JComboBox* fires an *ItemEvent*, so the event handler must implement the *ItemListener* interface.
- Adapter classes, which *implement* an interface and provide empty bodies for each method, are useful when only one of multiple component actions is processed.
- *MouseAdapter* and *MouseMotionAdapter* are adapter classes for the *MouseListener* and *MouseMotionListener* interfaces, respectively.
- The *GridLayout* layout manager arranges components into equally sized cells in rows and columns.
- A *BorderLayout* layout manager, which is the default layout manager for the *JFrame* class, organizes a container into five areas: north, south, west, east, and center, with each area holding at most one component. The size of each area expands or contracts depending on the size of the component in that area, the sizes of the components in the other areas, and whether the other areas contain a component.
- A *JPanel* component can be used as a general-purpose container. To create a complex arrangement of GUI components, we place several components into a panel and place the panel into the content pane of the current *JFrame*. Each panel and the content pane has its own layout manager.

16.17 Exercises, Problems, and Projects

16.17.1 Multiple Choice Exercises

1. An example of a GUI component class is

    - *ActionEvent*.
    - *actionPerformed*.
    - *JTextField*.
    - *ActionListener*.

6. What are the primary uses of GUI components? (Check all that apply.)

    - to display information
    - to facilitate the coding of methods
    - to let the user control the program
    - to collect information from the user

11. In what package do you find the *JButton*, *JTextField*, and *JComboBox* classes?

    - *javax.swing*
    - *java.swing*
    - *javax.awt*
    - *java.awt*
    - *java.io*

17. Components can be hidden.

    - true
    - false

20. In order to process an event when the user clicks on a button, what should the programmer do? (Check all that apply.)

    - Code a class that implements the *ActionListener* interface.
    - Declare and instantiate an object reference (a listener) of the class above.
    - Call the *actionPerformed* method.
    - Register the listener on the button.

25. Assuming everything has been coded correctly, what happens when the user clicks a button?

    - The *actionPerformed* method executes.
    - The *JButton* constructor executes.
    - The *main* method executes.

29. If you visit Oracle's Java website (*www.oracle.com/technetwork/java*) and look at the *KeyListener* interface, you will find that it has three methods: *keyPressed*, *keyTyped*, and *keyReleased*. We want to build a class that implements *KeyListener*. Which of the three methods should we implement?

    - *keyPressed* only
    - *keyReleased* only
    - *keyTyped* only
    - All three methods

34. You are designing a GUI with three buttons; a different action will be taken depending on which button the user clicks. You want to code only one *private* class implementing the *ActionListener* interface. Inside the *actionPerformed* method, which method do you call to determine which button was clicked?

    - *getButton*
    - *getSource*
    - *getOrigin*

38. A class extending the *JFrame* class can also implement a listener interface.

    - true
    - false

41. *NewLayout* is a layout manager.

    - true

- false

44. In the following code:

```
GridLayout gl = new GridLayout( 6, 4 );
```

Description
what do the arguments 6 and 4 specify?

- 6 refers to the number of columns and 4 to the number of rows.
- 4 refers to the number of columns and 6 to the number of rows.
- There will be 6 components organized in 4 different areas.
- There will be 4 components organized in 6 different areas.

49. What is the maximum number of components that a *BorderLayout* can manage?

- 2
- 3
- 4
- 5
- 6

55. In the following code:

```
BorderLayout bl = new BorderLayout( 7, 3 );
```

Description
what do the arguments 7 and 3 represent?

- the horizontal and vertical gaps between the five areas of the component
- the vertical and horizontal gaps between the five areas of the component
- the number of rows and columns in the component
- the number of columns and rows in the component

60. In the following code:

    ```
    contents.add( button, BorderLayout.NORTH );
    ```

    Description
    what is the data type of the second argument?

    - *int*
    - *String*
    - *BorderLayout*
    - *NORTH*

65. Components can be nested.

    - true
    - false

16.17.2 Reading and Understanding Code

For Questions 16 to 21, consider the following code:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class Game extends JFrame
{
 private JButton b1, b2, b3, b4;
 private Container contents;
 public Game( )
 {
  super( "Play this game" );
  contents = getContentPane( );
  contents.setLayout( new GridLayout( 2, 2 ) );
  b1 = new JButton( "Button 1" );
  b2 = new JButton( "Button 2" );
  b3 = new JButton( "Button 3" );
  b4 = new JButton( "Button 4" );
  contents.add( b1 );
  contents.add( b2 );
  contents.add( b3 );

  MyHandler mh = new MyHandler( );
  b1.addActionListener( mh );
  b2.addActionListener( mh );

  setSize( 400, 400 );
  setVisible( true );
 }

 private class MyHandler implements ActionListener
 {
  public void actionPerformed( ActionEvent ae )
  {
   System.out.println( "Hello" );
   if ( ae.getSource( ) == b2 )
      System.out.println( "Hello again" );
  }
 }
 public static void main( String [ ] args )
 {
  Game g = new Game( );
  g.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

Description

1. How many buttons will be displayed in the window?
2. How are the buttons organized in the window?
3. What is the text in the title bar of the window?
4. What happens when the user clicks on the button that says “Button 1”?
5. What happens when the user clicks on the button that says “Button 2”?
6. What happens when the user clicks on the button that says “Button 3”?

For Questions 22 to 26, consider the following code:

```
 import javax.swing.*;
 import java.awt.*;
 import java.awt.event.*;

 public class Game extends JFrame
 {
  private JCheckBox c1, c2, c3;
 private int value1, value2, value3;
 private Container contents;
 public Game( )
 {
  super( "Play this game" );
  contents = getContentPane( );
  contents.setLayout( new FlowLayout( ) );
  c1 = new JCheckBox( "Choice 1" );
  c2 = new JCheckBox( "Choice 2" );
  c3 = new JCheckBox( "Choice 3" );
  contents.add( c1 );
  contents.add( c2 );
  contents.add( c3 );
  MyHandler mh = new MyHandler( );
  c1.addItemListener( mh );
  c2.addItemListener( mh );
  c3.addItemListener( mh );

  setSize( 400, 400 );
  setVisible( true );
 }

 private class MyHandler implements ItemListener
 {
  public void itemStateChanged( ItemEvent ie )
  {
    if ( ie.getSource( ) == c1 )
    {
      if ( ie.getStateChange( ) == ItemEvent.SELECTED )
        value1 = 1;
      else
        value1 = 0;
    }
    else if ( ie.getSource( ) == c2 )
    {
     if ( ie.getStateChange( ) == ItemEvent.SELECTED )
       value2 = 2;
     else
       value2 = 0;
    }
    else if ( ie.getSource( ) == c3 )
    {
      if ( ie.getStateChange( ) == ItemEvent.SELECTED )
        value3 = 4;
      else
        value3 = 0;
   }
   System.out.println( ( value1 + value2 + value3 ) );
  }
 }
 public static void main( String [ ] args )
 {
   Game g = new Game( );
   g.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

Description

1. How many checkboxes will be displayed in the window?
2. How are the checkboxes organized in the window?
3. What happens when the user checks “Choice 3” only?
4. What happens when the user checks “Choice 1”, then checks “Choice 3”?
5. What happens when the user checks “Choice 1”, then “Choice 2”, then “Choice 3”?

16.17.3 Fill In the Code

For Questions 27 to 32, consider the following class:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class A extends JFrame
{
 private Container c;
 private JButton b;
 private JTextField tf;

}
```

Description

1. Inside the constructor, this code assigns the content pane of the frame to the *Container c*:

```
  // your code goes here
```

Description

2. Inside the constructor, this code instantiates the button *b* with the text “Button”:

```
// your code goes here
```

Description

3. Inside the constructor, this code instantiates the text field *tf*; after instantiation, the text field should be empty, but have space for 10 characters:

```
// your code goes here
```

Description

4. Inside the constructor, and assuming that *c* has been assigned the content pane, this code sets the layout manager of the content pane to a 2-by-1 grid layout manager:

```
// your code goes here
```

Description

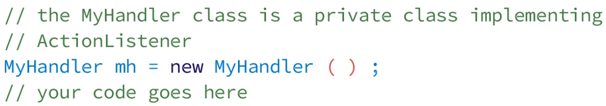
5. Inside the *actionPerformed* method of a *private* inner class implementing the *ActionListener* interface, this code changes the text of *tf* to “Button clicked” if the button *b* was clicked; otherwise, nothing happens:

```
public void actionPerformed ( ActionEvent ae )
{
// your code goes here

}
```

Description

6. Inside the constructor, this code registers the listener *mh*

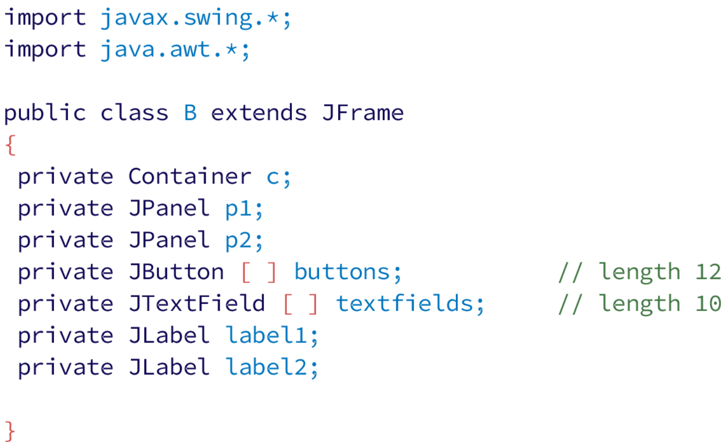
on the button *b*:



```
// the MyHandler class is a private class implementing
// ActionListener
MyHandler mh = new MyHandler ( ) ;
// your code goes here
```

Description

For Questions 33 to 41, consider the following class:



```
import javax.swing.*;
import java.awt.*;

public class B extends JFrame
{
 private Container c;
 private JPanel p1;
 private JPanel p2;
 private JButton [ ] buttons;          // length 12
 private JTextField [ ] textfields;    // length 10
 private JLabel label1;
 private JLabel label2;

}
```

Description

Also, assume that all the instance variables have been instantiated and you are coding inside the constructor.

1. This code sets the layout manager of the content pane to be a border layout manager.

```
// your code goes here
```

Description

2. This code adds *label1* to the north area of the window.

```
// your code goes here
```

Description

3. This code adds *label2* to the east area of the window.

```
// your code goes here
```

   Description

4. This code sets the layout manager of the panel *p1* to be a 3-by-4 grid layout manager.

```
// your code goes here
```

   Description

5. This code places all the buttons of the array buttons inside the panel *p1*.

```
// your code goes here
```

   Description

6. This code adds the panel *p1* in the center area of the window.

```
// your code goes here
```

   Description

7. This code sets the layout manager of the panel *p2* to a 5-by-2 grid with four pixels between cells, both horizontally and vertically.

```
// your code goes here
```

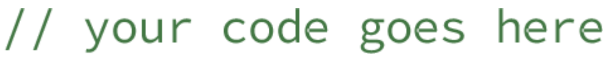
   Description

8. This code places all the text fields of the array *textfields* inside the panel *p2*.

```
// your code goes here
```

Description

9. This code adds the panel *p2* in the west area of the window.

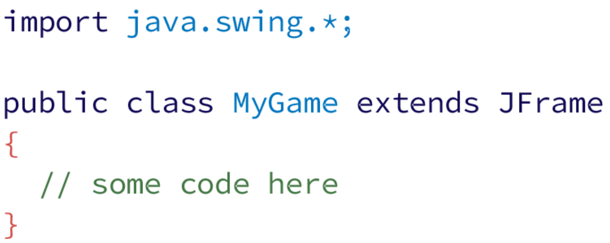


```
// your code goes here
```

Description

16.17.4 Identifying Errors in Code

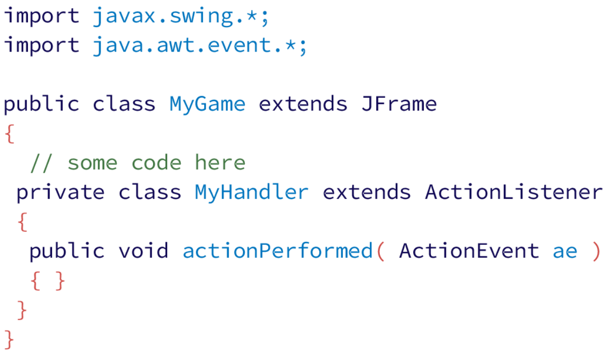
1. Where is the error in this code sequence?



```
import java.swing.*;

public class MyGame extends JFrame
{
   // some code here
}
```

Description

2. Where is the error in this code sequence?



```
import javax.swing.*;
import java.awt.event.*;

public class MyGame extends JFrame
{
   // some code here
 private class MyHandler extends ActionListener
 {
  public void actionPerformed( ActionEvent ae )
  { }
 }
}
```

Description

3. Where is the error in this code sequence?

```
import javax.swing.*;
import java.awt.event.*;

public class MyGame extends JFrame
{
 // some code here
 private class MyHandler implements ItemListener
 {
  public void actionPerformed( ActionEvent ae )
  { }
 }
}
```

Description

4. Where is the error in this code sequence?

```
import javax.swing.*;
import java.awt.event.*;

public class MyGame extends JFrame
{
 // some code here
 private class MyHandler implements ItemListener
 {
  public void itemStateChanged( ActionEvent e )
  { }
 }
}
```

Description

16.17.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. You coded the following in the file *MyGame.java*:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MyGame extends JFrame
{
 private Container c;
 private JLabel l;

 public MyGame( )
 {
  super( "My Game" );
  c = getContentPane( );
  c.setLayout( new FlowLayout( ) );
  c.add( l );  // Line 15
 }

 public static void main( String [ ] args )
 {
  MyGame mg = new MyGame( ); // Line 20
 }
}
```

Description

The code compiles, but at run time, you get the following output:

```
Exception in thread "main" java.lang.NullPointerException:
Cannot read field "parent" because "comp" is null
           at java.awt.Container.addImpl(Container.Java:1121)
           at java.awt.Container.add(Container.Java:440)
           at MyGame.<init>(MyGame.java:15)
           at MyGame.main(MyGame.java:20)
```

Description

Explain what the problem is and how to fix it.

2. You coded the following in the file *MyGame.java*:

```
import javax.swing.*;

public class MyGame extends JFrame
{
    public MyGame( )
    {
     super( "My Game" );
     setSize( 400, 400 );
    }
    public static void main( String [ ] args )
    {
     MyGame mg = new MyGame( );
    }
}
```

Description

The code compiles, but at run time, you cannot see the window and the code terminates.

Explain what the problem is and how to fix it.

3. You coded the following in the file *MyGame.java*:

```
import javax.swing.*;
import java.awt.*;

public class MyGame extends JFrame
{
 private Container c;
 private JTextField tf;

 public MyGame( )
 {
  super( "My Game" );
  c = getContentPane( );
  tf = new JTextField( "Hello" );
  c.add( tf, NORTH );
  setSize( 400, 400 );
  setVisible( true );
 }
 public static void main( String [ ] args )
 {
  MyGame mg = new MyGame( );
 }
}
```

Description

When you compile your code, you get the following error:

```
MyGame.java:14: error: cannot find symbol
     c.add( tf, NORTH );
                ^
   symbol:   variable NORTH
   location: class MyGame
1 error
```

Description

Explain what the problem is and how to fix it.

4. You coded the following in the file *MyGame.java*:

```
import javax.swing.*;
import java.awt.*;

public class MyGame extends JFrame
{
    // Some code here
    private class MyHandler implements ActionListener
    {
     public void actionPerformed( ActionEvent ae )
     { }
    }
}
```

Description

When you compile your code, you get the following error:

```
MyGame.java:7: error: cannot find symbol
     private class MyHandler implements ActionListener
                                       ^
   symbol: class ActionListener
   location: class MyGame
MyGame.java:9: error: cannot find symbol
     public void actionPerformed( ActionEvent ae )

     symbol: class ActionEvent
     location: class MyGame.MyHandler
  2 errors
```

Description

Explain what the problem is and how to fix it.

5. You coded the following in the *MyGame.java* file:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MyGame extends JFrame
{
 private JButton b;
 private JTextField tf;
 private Container contents;
 public MyGame( )
 {
  super( "Play this game" );
  contents = getContentPane( );
  contents.setLayout( new GridLayout( 1, 2 ) );
  b = new JButton( "Click here" );
  tf = new JTextField( "Hello" );
  contents.add( b );
  contents.add( tf );
  setSize( 400, 400 );
  setVisible( true );
 }

 private class MyHandler implements ActionListener
 {
  public void actionPerformed( ActionEvent ae )
  {
   tf.setText( "Hi" );
  }
 }
 public static void main( String [ ] args )
 {
   MyGame g = new MyGame( );
   g.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

Description

The code compiles and runs. However, when you click the button, the text in the text field does not change.

Explain what the problem is and how to fix it.

16.17.6 Write a Short Program

1. Write a program that displays a text field and two buttons labeled “uppercase” and “lowercase”. When the user clicks on the uppercase button, the text changes to uppercase; when the user clicks on the lowercase button, the text changes to lowercase.

2. Write a program with two radio buttons and a text field. When the user clicks on one radio button, the text changes to lowercase; when the user clicks on the other radio button, the text changes to uppercase.
3. Write a program with two checkboxes and a text field. When no checkbox is selected, no text shows in the text field. When only the first checkbox is selected, the word “hello” shows in lowercase. When only the second checkbox is selected, the word “HELLO” shows in uppercase. When both checkboxes are selected, the word “HeLlO” shows (lower- and uppercase letters alternate).
4. Write a program with three radio buttons and a circle. (You can choose whether you draw the circle or if the circle is a label image.) When the user clicks on the first radio button, the circle becomes red. When the user clicks on the second radio button, the circle turns orange. When the user clicks on the third radio button, the circle becomes blue.
5. Write a program with three checkboxes and a circle drawn in black. Like the checkbox example in the chapter, each checkbox represents a color (red, blue, or green). Depending on the checkboxes selected, compute the resulting color as follows. If a checkbox is selected, assign the value 255 to the amount of the corresponding color. If a checkbox is not selected, assign 0 to the amount of the corresponding color. For example, if the checkboxes representing the colors red and blue are selected, the resulting color should be *Color (255, 0, 255*). Color the circle appropriately.
6. Write a program that simulates a guessing game in a GUI program. Ask the user for a number between 1 and 6 in a text field, then roll a die randomly and tell the user if he or

she won. Write the program in such a way that any invalid user input (i.e., not an integer between 1 and 6) is rejected and the user is asked again for input.

7. Write a program that simulates a guessing game in a GUI program. Generate a random number between 1 and 100; that number is hidden from the user. Ask the user for a number between 1 and 100 in a text field, then tell the user whether the number is too high, too low, or the correct number. Let the user continue guessing until he or she guesses the correct number.
8. Modify the *JComboBox* example in this chapter to use *JList* instead.

9. Write a program that displays a 4-by-6 grid of buttons, each with some unique text. One button is the “winning” button, which your program determines randomly, inside the constructor. When the user clicks on a button, its text is changed to “No” if the button clicked is not the winning button, or to “Won” if the button clicked is the winning

button.

10. Same as Exercise 59 with the following additions: Keep track of how many times the user clicks on buttons. If the user has not won after five clicks, the text on the last button clicked should be changed to “Lost”. Once the user has lost or won, you should disable the game—that is, the buttons no longer respond to clicks from the user.
11. Write a program that displays in the title of the window the position of the mouse as the user moves the mouse around the window.
12. Write a program that draws a small circle that follows the mouse as the user moves the mouse around the window.

16.17.7 Programming Projects

1. Write a GUI-based tic-tac-toe game for two players.
2. Write a GUI-based program that analyzes a word. The user will type the word in a text field. Provide buttons for the following:

    - One button, when clicked, displays the length of the word.
    - Another button, when clicked, displays the number of vowels in the word.
    - Another button, when clicked, displays the number of uppercase letters in the word.

    For this, you should design and code a separate (non-GUI) class encapsulating a word and its analysis, then instantiate an object of that class inside your GUI class and call the various methods as needed.

6. Write a GUI-based program that analyzes a soccer game. The user will type the names of two teams and the score of the game in four text fields. You should add appropriate

labels and create buttons for the following:

- One button, when clicked, displays which team won the game.
- Another button, when clicked, displays the game score.
- Another button, when clicked, displays by how many goals the winning team won.

For this, you should design and code a separate (non-GUI) class encapsulating a soccer game, then instantiate an object of that class inside your GUI class and call the various methods as needed.

10. Write a GUI-based program that analyzes a round of golf. You will retrieve the data for 18 holes from a text file. On each line in the file will be the par for that hole (3, 4, or 5) and your score for that hole. Your program should read the file and display a combo box listing the 18 holes. When the user selects a hole, the score for that hole should be displayed in a label. Provide buttons for the following:

    - One button, when clicked, displays whether your overall score was over par, under par, or par.
    - Another button, when clicked, displays the number of holes for which you made par.
    - Another button, when clicked, displays how many birdies you scored (a birdie on a hole is 1 under par).

    For this, you should design and code a separate (non-GUI) class encapsulating the analysis, then instantiate an object of that class inside your GUI class and call the various methods as needed.

14. Write a GUI-based program that analyzes statistics for tennis players. You will retrieve the data from a text file. On each line in the file will be the name of a player, the player’s number of wins for the year, and the player’s

number of losses for the year. Your program should read the file and display the list of players. When the user selects a player, the winning percentage of the player should be displayed in a label. Provide buttons for the scenarios that follow.

- One button, when clicked, displays which player had the most wins for the year.
- Another button, when clicked, displays which player had the highest winning percentage for the year.
- Another button, when clicked, displays how many players had a winning record for the year.

For this, you should design and code a separate (non-GUI) class encapsulating the tennis statistics analysis, then instantiate an object of that class inside your GUI class and call the various methods as needed.

18. Write a GUI-based program that simulates the selection of a basketball team. You will retrieve the data from a text file containing 10 lines. On each line will be the name of a player. Your program needs to read the file and display 10 checkboxes representing the 10 players. A text area will display the team, made up of the players being selected. A basketball team has five players. Your program should not allow the user to change his or her selection after the team has five players. Every time the user checks or unchecks a checkbox, the team in the text area should be updated accordingly. Provide buttons for the following:

    - One button, when clicked, displays how many players are currently on the team.
    - Another button, when clicked, displays how many players remain unselected.

    For this, you should design and code a separate (non-GUI) class encapsulating the player selection process, then instantiate an object of that class inside your GUI class and call the various methods as needed.

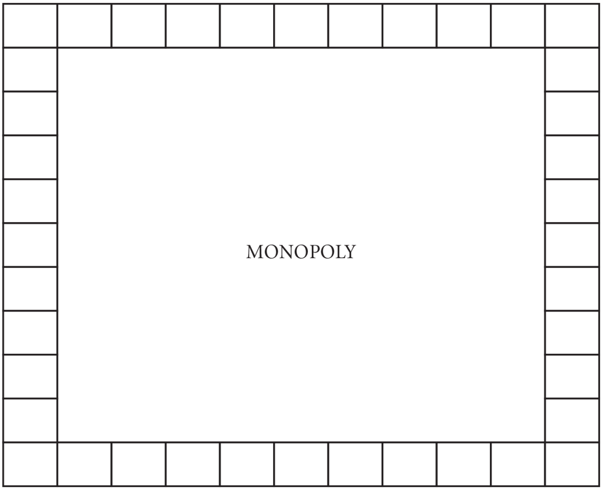
21. Write a GUI-based program that enables the user to choose a file containing an image. Your application then displays that image in a label. Your GUI should include a button and a label. When the user clicks on the button, open a file-choosing dialog box to enable the user to select a file (Hint: look up the *JFileChooser* class in the Java Class Library). The dialog box should show the files in the current directory with an extension of either *jpg* or *gif*.

22. Write a GUI-based program that displays a team on a soccer field. You will retrieve the data from a text file containing 11 lines. Each line will contain the name of a player. Your program should read the file and display the following window when it starts (you can assume that the players in the file are not in any particular order). Each cell is a button; when the user clicks on a button, the button replaces its text with the name of the player.

Left wing (11)	Striker (9)	Right wing (7)
Left midfielder (6)	Midfielder (10)	Right midfielder (8)
Left defender (3)	Stopper (4)	Right defender (2)
	Sweeper (5)	
	Goalie (1)	

---

1. Write a GUI-based program that displays a Monopoly® game. Add labels for the four train stations. Add buttons for all the Chance cells and set the text of these to a question mark. When the user clicks on one of the buttons, set its text to a message of your choice, chosen randomly from four messages.



Description

2. Write a GUI-based, simple drawing program. This program should have two buttons: one allowing the user to draw a rectangle, the other allowing the user to draw an oval. The user draws either a rectangle or an oval by pressing the mouse, dragging it, and releasing it. The top-left (*x*, *y*) coordinate of the rectangle (or enclosing rectangle for the oval) is where the user pressed the mouse button; the bottom-right point of the rectangle (or enclosing rectangle for the oval) drawn is where the user released the mouse

button.You will need to organize the window in two areas: one for the buttons, one for drawing.

16.17.8 Technical Writing

1. You are writing a program that you expect to be used by many users, with a different computer system. Would you use layout managers or would you hard code the position of components inside your GUI? Discuss.

16.17.9 Group Project (for a group of 1, 2, or 3 students)

1. Design and code a program that simulates an auction. You should consider the following:

   A file contains a list of items to be auctioned. You can decide on the format of this file and its contents. For example, the file could look like this:

   ```
   Oldsmobile, oldsmobile.gif, 100
   World Cup soccer ticket, soccerTickets.gif, 50
   Trip for 2 to Rome, trip.gif, 100
   ```

   In the preceding file sample, each line represents an item as follows: the first field is the item's description, the second field is the name of a file containing an image of the item, and the third field is the minimum bid. You can assume that each item's description is unique.

   Items are offered via an online-like auction. (You do not need to include any network programming; your program is a single-computer program.) Users of the program can choose which item to bid on from a list or a combo box. Along with displaying the description of the item, your program should show a picture of the item and the current highest bid (at the beginning, the current highest bid is the minimum bid). Users bid on an item by selecting the item, typing their name (you can assume that a different name will be used each time), and entering a price for the item. Your program should remember the highest bidder and the highest bid for each item by writing the information to a file.

Furthermore, each time a bid is made, the item’s highest bid, displayed on the screen, should be updated if necessary.

© Lois GoBe/Shutterstock

Chapter 17

Graphical User Interfaces Using JavaFX

CHAPTER CONTENTS

Introduction
17.1 The Structure of a JavaFX Application
17.2 GUI Controls
17.3 A Simple Control: *Label*
17.4 Event Handling: Managing User Interactions
17.5 Text Fields and Command Buttons
17.6 Radio Buttons and Checkboxes
17.7 **Programming Activity 1: Working with Buttons**
17.8 Combo Boxes
17.9 Sliders
17.10 Building a GUI Programmatically
17.11 Layout Containers: Dynamically Setting Up the GUI Using *GridPane*
17.12 *BorderPane* Layout, Animations, Sounds, and Lambda Expressions
17.13 Nesting Components
17.14 Mouse and Touch Events
17.15 Using a List to Display a Pie Chart

17.16 Using a List to Display a Dynamic Bar Chart
17.17 Using a Style Sheet to Style the View
17.18 **Programming Activity 2: Working with Layout Containers**
17.19 Chapter Summary
17.20 Exercises, Problems, and Projects

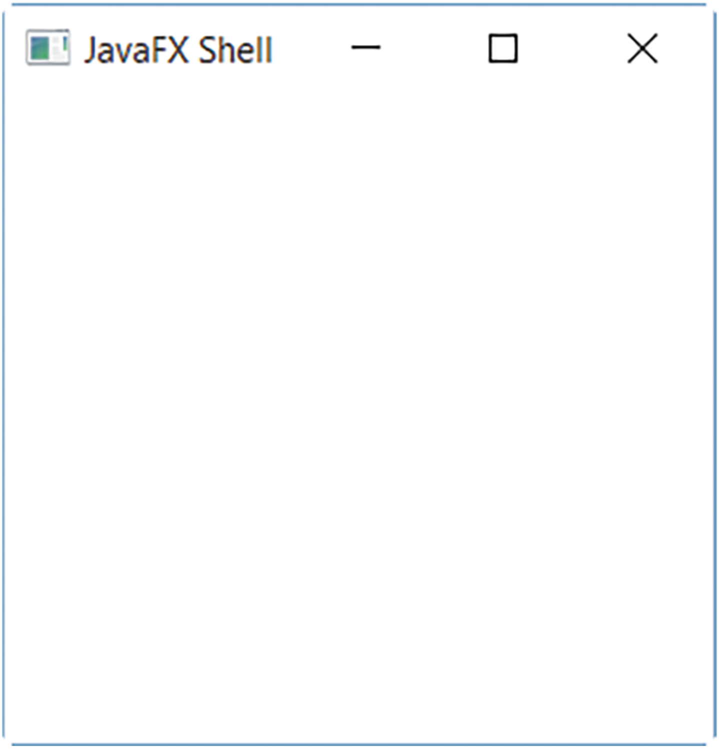
- 17.20.1 Multiple Choice Exercises
- 17.20.2 Reading and Understanding Code
- 17.20.3 Fill In the Code
- 17.20.4 Identifying Errors in Code
- 17.20.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
- 17.20.6 Write a Short Program
- 17.20.7 Programming Projects
- 17.20.8 Technical Writing
- 17.20.9 Group Project

Introduction

Many applications we use every day have a Graphical User Interface, or GUI (pronounced Goo-ey). These GUIs allow the user to communicate to the application by entering text into boxes; pressing buttons; or selecting items from a list, a set of radio buttons, or checkboxes.

GUIs allow the user to drive the application by selecting the next function to be performed, entering the needed data, or setting program preferences, such as colors or fonts. Applications with GUIs are usually easier to learn and use because the interface is familiar to the user.

One way to create a GUI is to use the Swing components. Recently, Oracle introduced JavaFX, a new approach to making GUIs. All the classes needed to create a JavaFX application are automatically included with Java SE; nothing more needs to be downloaded. In this chapter, we present some of the many JavaFX classes, along with the main concepts associated with developing a JavaFX application.



17.1 The Structure of a JavaFX Application

The top-level structure in a JavaFX application is the **stage**, which corresponds to a window. A stage can have one or more **scenes**, which are top-level containers for **nodes** that make up the window contents. A node can be a user interface control, such as a button or a drop-down list; a layout; an image or other media; a graphical shape; a web browser; a chart; or a group. In this chapter, we concentrate on user interface controls, layouts, charts, and images.

To create a JavaFX GUI, we add nodes to a scene. These nodes are arranged in a hierarchy, called a **scene graph**, in which some nodes are children of other nodes. The top node is called the **root**.

JavaFX applications can be built in several ways. If we know which controls our interface needs and how they should be arranged, we can use **FXML**, a scripting language based on **XML** (**Extensible Markup Language)**. In this case, we create an FXML file where we specify the layout container and nodes and their properties. For dynamic GUIs where the number or types of controls are determined at runtime, we can define the number, type, properties, and positioning of controls programmatically (that is, through Java code).

In this chapter, we start by using FXML and use Java code later. By defining our GUI using FXML, we separate the GUI from the logic of the code. Also, FXML allows nonprogrammers to contribute to a team by defining the GUI without knowledge of programming.

REFERENCE POINT

You can get more information about the JavaFX classes at Oracle's Java website: *www.oracle.com/technetwork/java.*

Examples 17.1 and 17.2 show the basic structure of a JavaFX application that uses FXML.

Example 17.1 A JavaFX Shell Application

```java
/*  JavaFX Shell Application
    Anderson, Franceschi
*/

import java.net.URL;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.layout.HBox;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class FXShellApplication extends Application
{
  @Override
  // start is main entry point for the application.
  // It receives a Stage object - the main window for the
  // GUI application
  public void start( Stage stage ) // throws Exception
  {
    try
    {
      // Locate the FXML resource
      URL url
         = getClass( ).getResource( "fxml_shell.fxml" );

      // Load the FXML resource, instantiate root Node;
      // use appropriate layout class for root Node;
      // here we use HBox
      HBox root = FXMLLoader.load( url );

      // create a scene associated with the root
      // and set its width and height
      Scene scene = new Scene( root, 300, 275 );

      // assign the scene to the stage object
      stage.setScene( scene );

      // set title of stage (optional)
      stage.setTitle( "JavaFX Shell" );

      // make the stage visible
      stage.show( );
    }
    // The FXMLLoader load method throws an exception if
    // the FXML file is invalid or the URL was not found
    catch ( Exception e )
    {
      e.printStackTrace( );
    }
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description

Lines 5 through 10 import the classes we need in most applications. Notice that the JavaFX classes are stored in packages having the prefix of *javafx*.

At line 12, we define our class as inheriting from the *abstract Application* class, which is the entry point for all JavaFX applications. When a JavaFX application is launched, the runtime first calls the *init* method, and then *start*. When the application closes its last window or if it calls the *Platform.exit* method, the runtime calls the *stop* method. The *Application* class provides concrete implementations of the *init* and *stop* methods, so overriding *init* and *stop* is optional. The *Application* class defines the *start* method as *abstract*, however, so we must override the *start* method. **Table 17.1** shows some methods of the *Application* class.

Some Methods of the *Application* Class

<table>
<tr><th>Package</th><th>javafx.application</th></tr>
<tr><th>Return value</th><th>Method name and argument list</th></tr>
<tr><td><code>void</code></td><td><code>init( )</code></td></tr>
<tr><td colspan="2">called by the JavaFX runtime when the application is launched. Overriding this method is optional.</td></tr>
<tr><td><code>void</code></td><td><code>launch(String … args)</code></td></tr>
<tr><td colspan="2"><i>static</i> method that launches the application. Passes any command-line arguments to the application.</td></tr>
<tr><td><code>Application.Parameters</code></td><td><code>getParameters( )</code></td></tr>
<tr><td colspan="2">returns any command-line arguments as an <i>Application.Parameters</i> object.</td></tr>
<tr><td><code>void</code></td><td><code>start(Stage primaryStage)</code></td></tr>
<tr><td colspan="2">the main entry point for JavaFX applications. The application places its scene onto the <i>primaryStage</i>. This <i>abstract</i> method needs to be implemented.</td></tr>
<tr><td><code>void</code></td><td><code>stop( )</code></td></tr>
<tr><td colspan="2">called when the application signals that it is ready to end. Overriding this method is optional.</td></tr>
</table>

Description

SOFTWARE ENGINEERING TIP

Using the *@Override* annotation whenever you intend to override an inherited method reduces errors because the compiler alerts you if your method header is not correct.

Lines 14 through 50 define the *start* method. In line 14, we use the *@Override* annotation. It is good practice to use this annotation whenever we intend to override an inherited method. Using the annotation causes the compiler to alert us if our method header does not correctly override the method in the superclass.

A lot is happening in lines 22 through 29. The FXML file contains the definitions for our GUI components. In order to associate the FXML file with the application, we need to create a URL object containing the file's location; then we load the file as an application resource. We inherit the *getClass* method from the *Object* class. It returns the *Class* object associated with our application. We then use that object reference to call the *getResource* method in the *Class* class, shown in **Table 17.2**.

The *getResource* Method of the *Class<T>* Class

Package	**java.lang**
Return value	**Method name and argument list**
`URL`	`getResource( String resource )`
returns a URL object representing the location of the *resource,* or *null* if the *resource* is not found	

Description

Once we have a URL for the FXML file, we call the *static load* method of the *FXMLLoader* class, shown in **Table 17.3**, to instantiate the layout and components that we defined in our

FXML file. This method throws an *IOException* if the format of the FXML file is invalid. In this example, the top-level node is an *HBox* layout object, which arranges its child nodes horizontally across the window. The root and its children make up the scene graph. The return value from the *load* method is a reference to the top-level, or root, node in the FXML file.

The *load* Method of the *FXMLLoader* Class

Package	**javafx.fxml**
Return value	**Method name and argument list**
`<T> T`	`load( URL location )`
static method that instantiates the nodes defined in the FXML file specified by *location* and returns a reference to the root node	

Description

After we have the resources loaded, we instantiate a *Scene* object (line 33), which is the container for our GUI content. **Table 17.4** shows a constructor of the *Scene* class. Some of the other *Scene* constructors do not require width and height parameters. With those constructors, the scene is sized to fit its contents. For this scene, we specify the layout node returned from the *load* method as the root, and we size the scene to be 300 pixels wide and 275 pixels high.

A Constructor of the *Scene* Class

Package	**javafx.scene**
`Scene( Parent root, double width, double height )`	
instantiates the *Scene* with a top-level node of *root* and sets the width and height to the pixel values specified	

Description

Now we are ready to set the stage, literally. Notice that the *start* method receives a *Stage* object as a parameter (line 18). This corresponds to the top-level window. At line 36, we allocate our newly created scene to the *stage* object. At line 39, we set the text to appear in the title bar, and at line 42, we make the window visible. **Table 17.5** shows some useful methods of the *Stage* class.

Useful Methods of the *Stage* Class

Package	**javafx.stage**
Return value	**Method name and argument list**
`void`	`setScene( Scene scene )`
specifies the scene to be hosted by the *Stage*.	
`void`	`setTitle( String title )`
sets the text to appear in the window title bar.	
`void`	`show( )`
makes the window visible. By default, windows are not visible.	

Description

We placed most of the code in the *start* method into a *try* block because several exceptions can occur. If the *getResource* method of the *Class* class does not find the FXML file, it returns *null.* This causes a *NullPointerException* when the *FXMLLoader* *load* method tries to access the file. The *load* method also throws an *IOException* if the FXML file format is invalid. To simplify the examples in this chapter, we use one *catch* block (lines 44–49) that specifies *Exception*, which is a superclass of both *IOException* and *NullPointerException*. In the *catch* block, we print the stack trace to help with debugging errors in our applications. It is good software engineering practice, however, in the final application to catch each exception separately and output a meaningful message to the user.

COMMON ERROR TRAP

Windows are hidden by default. Be sure to call the *show* method at the end of the *start* method. Otherwise, the window will not appear.

SOFTWARE ENGINEERING TIP

Catch each possible exception separately. If recovering from the exception is not possible, output a meaningful message to the user.

REFERENCE POINT

You can get more information about FXML at *http://docs.oracle.com/javafx/2/get_started/jfxpub-get_started.htm*. For more information about XML, visit *http://www.w3.org/XML.*

At lines 52 through 55, we include the *main* method. Its only job is to start the application by calling the *launch* method of the *Application* class, passing any parameters that have been sent to the application. An application may access these parameters using the *getParameters* method.

In **Example 17.2**, we show a shell FXML file. In this file, we use FXML to define the layouts and controls that make up the GUI.

Example 17.2 Shell FXML file: *fxml_shell.fxml*

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <!-- import classes -->
4 <?import javafx.scene.layout.HBox ?>
5
6 <!-- use appropriate layout manager -->
7 <HBox xmlns:fx="http://javafx.com/fxml" >
8    <!-- define GUI components here -->
9 </HBox>
```

Description

Line 1 is the header. Like other lines that do not specify XML elements, it begins with <? and ends with ?>. It defines the XML version as 1.0 and the encoding to be UTF-8, which is essentially Unicode, except that the first 128 characters are stored as 8 bits. Lines 3, 6, and 8 are comments, which use the same syntax as an HTML comment; that is, they begin with <!-- and end with -->. These comments can span more than one line.

COMMON ERROR TRAP

Be sure to include an *import* statement for any controls or layout containers referenced in the FXML file. Omitting an *import* statement will generate an exception when the FXML Loader loads the file.

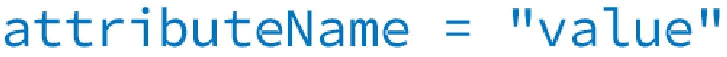
At line 4 we import the *HBox* class because it is the root element defined in this sample file. Similarly, we need to import any classes referenced in the FXML file.

At line 7, we have chosen the *HBox* layout class as an example. In other applications, we put the appropriate top-level layout class here, which becomes the root node for our layout. Lines 7 through 9 illustrate the syntax of an **element**, which we use to define the components of the GUI.

Elements begin with a **start tag** with the element’s name enclosed in angle brackets (< >). Some elements, such as the *HBox* definition here, have **closing tags**, which are simply the element’s name preceded by a forward slash (/), also enclosed in angle brackets (line 9). Some tags, illustrated later in this chapter, have an empty closing tag and end with />.

Elements can have **attributes**, which further define the element or set properties of the element. Attribute definitions are inserted between the element’s name and its closing tag.

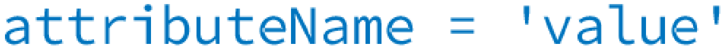
The syntax for a JavaFX attribute is



```
attributeName = "value"
```

Description

or



```
attributeName = 'value'
```

Description

COMMON ERROR TRAP

Be sure to enclose attribute values in single or double quotes. Omitting one or both quotes will generate an exception when the FXML Loader loads the file.

At line 7, we define an attribute for the root layout:

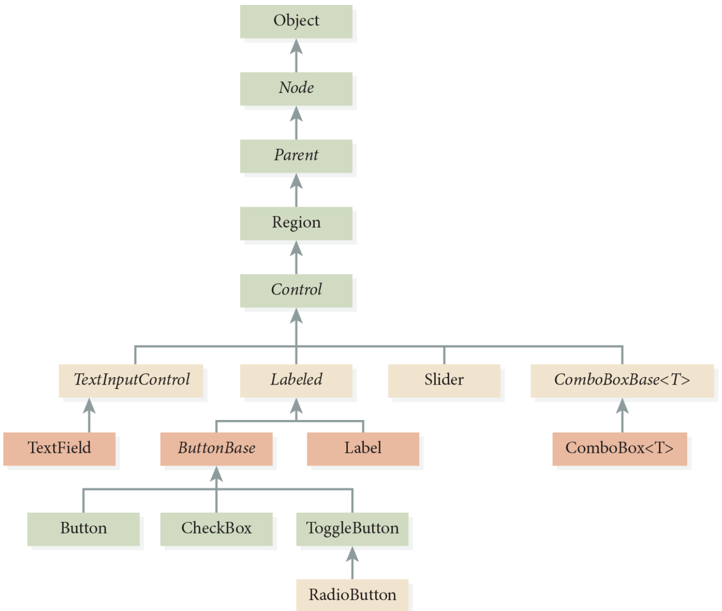
```
xmlns:fx="http://javafx.com/fxml"
```

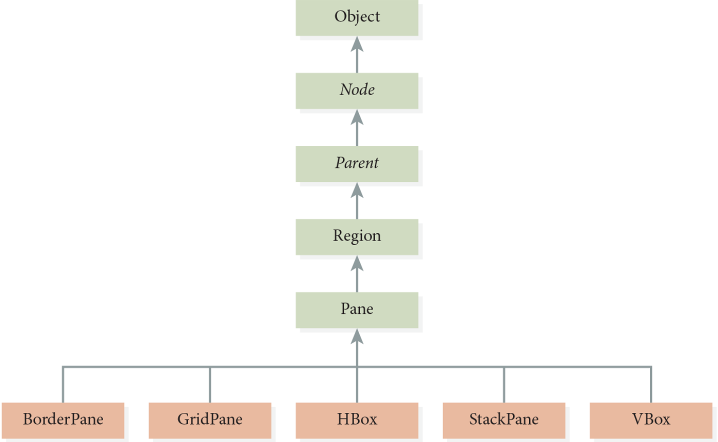
Description

This attribute defines the **namespace** for FXML elements. A namespace prevents duplication of variable names, called *name collisions*, by defining a scope for variable names.

Figure 17.1 shows the window created when we run *FXShellApplication.java.* Note that the text in the title bar is "JavaFX Shell" as set in Example 17.1, line 39. The window is empty because we have not added any controls. We add some content to a window in the next section.

Figure 17.1 The Window Created by *FXShellApplication*


Object
Node
Parent
Region
Control
TextInputControl
Labeled
Slider
ComboBoxBase<T>
TextField
ButtonBase
Label
ComboBox<T>
Button
CheckBox
ToggleButton
RadioButton


Object
Node
Parent
Region
Pane
BorderPane
GridPane
HBox
StackPane
VBox

17.2 GUI Controls

JavaFX provides an extensive set of classes that can be used to add a GUI to our applications. A GUI control performs at least one of these functions:

- Displays information
- Collects data from the user

- Allows the user to initiate program functions

Table 17.6 lists some JavaFX classes that encapsulate GUI controls. All classes listed in Table 17.6 belong to the package *javafx.scene.control*.

Selected GUI Controls and Their JavaFX Classes

Package	**javafx.scene.control**
JavaFX Class	**Purpose**
Label	Displays an image or read-only text. Labels are often used to identify the contents of *TextFields.*
TextField	A single-line text box for accepting user input.
Button	Command button that the user clicks to signal that an operation should be performed.
RadioButton	Toggle button that the user clicks to select one option in a group.
CheckBox	Toggle button that the user clicks to select or deselect 0, 1, or more options in a group.
ComboBox	List of options from which the user selects one item.
Slider	Displays a set of continuous values along a horizontal or vertical line. The user can select a value by moving the knob, or thumb.

Description

Figure 17.2 shows the hierarchy of some JavaFX controls. Recall that because of the “is a” relationship in inheritance, a subclass object is also an object of each of its superclasses. Thus, all the controls are a *Node* (an element of the scene), a *Parent (*a *Node* that can have children controls), a *Region* (a resizable container that can be styled), and a *Control* (a GUI component). Along the hierarchy, each control has gained methods and properties from its superclasses.

Figure 17.2 The Hierarchy of Some JavaFX Controls

Description

In our applications, we place the controls into the scene by using layout containers, which organize the controls according to each layout's rules. Some of these layout containers are shown in **Table 17.7.** These classes are in the *javafx.scene.layout* package.

Commonly Used JavaFX Layout Classes

Package	**javafx.scene.layout**
Layout Container	**Lays out its children nodes . . .**
HBox	In a single horizontal row.
VBox	In a single vertical column.
BorderPane	With at most one child in its top, left, right, bottom, and center positions.
GridPane	In a grid of rows and columns. A child can span more than one row or column.
StackPane	In a front-to-back stack.

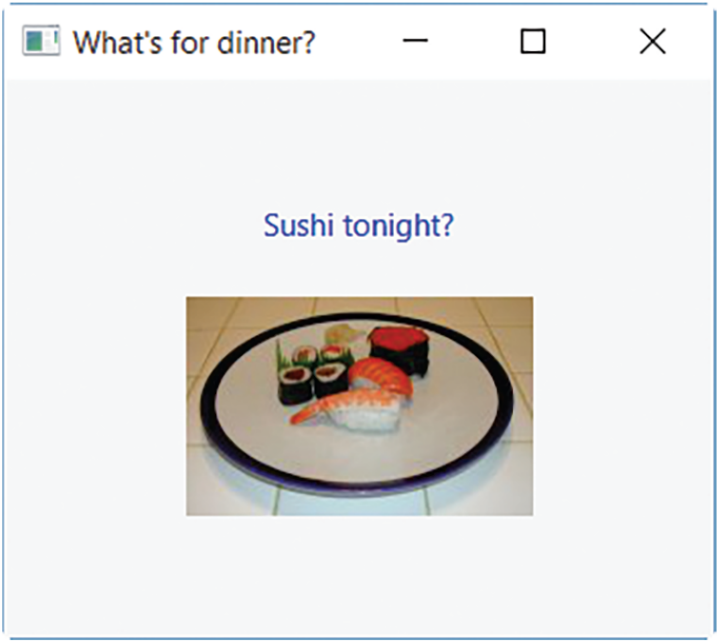
Description

The hierarchy of the layout classes is shown in **Figure 17.3**. Like the controls, the layout classes also inherit from *Node*, *Parent,* and *Region*. Instead of inheriting from *Control*, however, these classes inherit from *Pane,* which provides a method for accessing all the children in the scene graph. We use that method later in this chapter.

Figure 17.3 The Layout Classes Hierarchy

Description

In the examples in this chapter, we set the root node of our scenes to be one of these layouts and add controls, images, and even other layouts as child nodes. We show how to nest layouts in later examples in this chapter.



Sushi tonight?

17.3 A Simple Control: *Label*

Starting with our shell JavaFX application, let's add a simple control to the window, a *Label,* and an image, which we display through an *ImageView* node. **Example 17.3** shows the application code. Notice that the code is almost identical to Example 17.1, except for the name of the FXML file (line 22), the title bar text (line 32), and that the *root* node is a *VBox* layout (line 24).

Example 17.3 *Dinner.java*

```
/*  Displaying a Label and image
    Anderson, Franceschi
*/

import java.net.URL;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.layout.VBox;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Dinner extends Application
{

  @Override
  public void start( Stage stage )
  {
    try
    {
      // find the XML resource
      URL url
         = getClass( ).getResource( "fxml_dinner.fxml" );
      // load the XML resource and instantiate the root node
      VBox root = FXMLLoader.load( url );

      // create a scene
      Scene scene = new Scene( root, 350, 275 );

      // set the scene
      stage.setScene( scene );
      // set title of stage
      stage.setTitle( "What's for dinner?" );
      // show the stage
      stage.show( );
    }
    catch ( Exception e )
    {
      e.printStackTrace( );
    }
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description

Again, we have an accompanying FXML file, shown in **Example 17.4.** We begin this file like the shell FXML file, with the definition of the FXML version and encoding scheme (line 1);

then we import all the classes we reference (lines 3–5). In this file, we define the layout to be a *VBox* (lines 7–12). We use attributes to assign values to properties of the *VBox* layout (line 7); we set the *alignment* property to center the child nodes in the window and the *spacing* property to insert 25 pixels between the nodes in the layout.

Example 17.4 The *fxml_dinner.fxml* File

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 
 3 <?import javafx.scene.control.*?>
 4 <?import javafx.scene.image.*?>
 5 <?import javafx.scene.layout.*?>
 6 
 7 <VBox id="root" alignment="CENTER" spacing="25" >
 8   <Label text="Sushi tonight?" textFill="BLUE" />
 9   <ImageView>
10      <Image url="@sushi.jpg" />
11   </ImageView>
12 </VBox>
```

Description

At line 8, we define a *Label* control and use attributes to set its text to “Sushi tonight?” and its text color (*textFill*) to blue. All attribute values must be plain text, enclosed in double quotes. If an object or primitive value is needed, the FXML Loader performs any necessary conversion. Thus for the text color, we specify the attribute value as “BLUE,” and the FXML Loader converts the text to the object *Color.BLUE*.

REFERENCE POINT

To find the properties that can be set for any layout or control, see the JavaFX class documentation. Remember also that controls inherit properties from their superclasses.

At lines 9 through 11, we define a second child node as an *ImageView* element. We set the *Image* property of the *ImageView* to the file “sushi.jpg” (line 10). We preface the filename with “@” to indicate that we are specifying the filename relative to the current folder. Thus, the file *sushi.jpg* is stored in the same folder as the *.java* file. When this application runs, we see the window shown in **Figure 17.4**.

Figure 17.4 The Window Produced by Examples 17.3 and 17.4

Description

with these end-of-chapter questions

17.20.1	Multiple Choice Exercises Questions 1, 2, 3, 4, 5
17.20.4	Identifying Errors in Code Question 53
17.20.5	Debugging Area—Using Messages from the Java Compiler and Java JVM Questions 59, 60, 62

17.4 Event Handling: Managing User Interactions

Now we know how to open a window in an application, and we know how to display a label and an image. The user, however, cannot yet interact with our application. We need to add some interesting GUI controls, such as text entry fields and a button. By interacting with those controls, the user can enter data and initiate operations in our programs.

GUI programming uses an **event-driven model**, as opposed to the procedural model of programming that we have used thus far. By event-driven, we mean that by using a GUI, we put the user in control of what happens next. For example, we might display some text entry fields, some buttons, and a selectable list of items. Then our program "sits back" and waits for the user to interact with the controls. When the user presses a button or selects an item from the list, our application responds by performing the operation the user requested. Then the application sits back again and waits for the user to press another button or select another item from the list. These user actions generate **events**. Thus, the processing of our application consists of responding to events caused by the user interacting with our GUI controls.

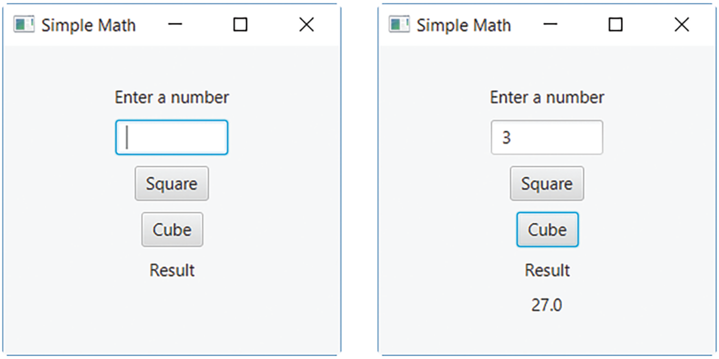
When the user interacts with a GUI control, the control **fires an event.** To handle that event, we register our application's interest in being notified when a particular event occurs, and we provide code—an **event handler**, also called a **listener**—to execute when the event occurs**.**

JavaFX supports and encourages the **Model-View-Controller** architecture for writing GUI applications. In this architecture,

- The **Model** manages the data of the application and its state.
- The **View** presents the user interface.
- The **Controller** handles events generated by the user and

communicates those changes to the Model, which updates its state accordingly and communicates any changes back to the Controller. The Controller then updates the View to reflect those changes.

These three components can be placed in the same or in different files.


Simple Math
Enter a number
Square
Cube
Result
Simple Math
Enter a number
3
Square
Cube
Result
27.0

17.5 Text Fields and Command Buttons

Let's illustrate event handling and the Model-View-Controller architecture with an example that allows a user to square or cube a number. We provide a *TextField* for entering the number; two *Button*s, one for squaring the number and the other for cubing the number; and *Labels* for displaying a prompt and for displaying the squared or cubed result.

The interface we want to create is shown in **Figure 17.5A**. **Figure 17.5B** shows the window after the user entered "3" into the *TextField* and clicked the "Cube" button.

Figure 17.5 (A) The Initial Window (B) The Window After User Interaction

Description

For this application, our Model, shown in **Example 17.5**, is minimal. It provides two *static* methods, *square* (lines 7–14) and *cube* (lines 16–23), to perform the calculations.

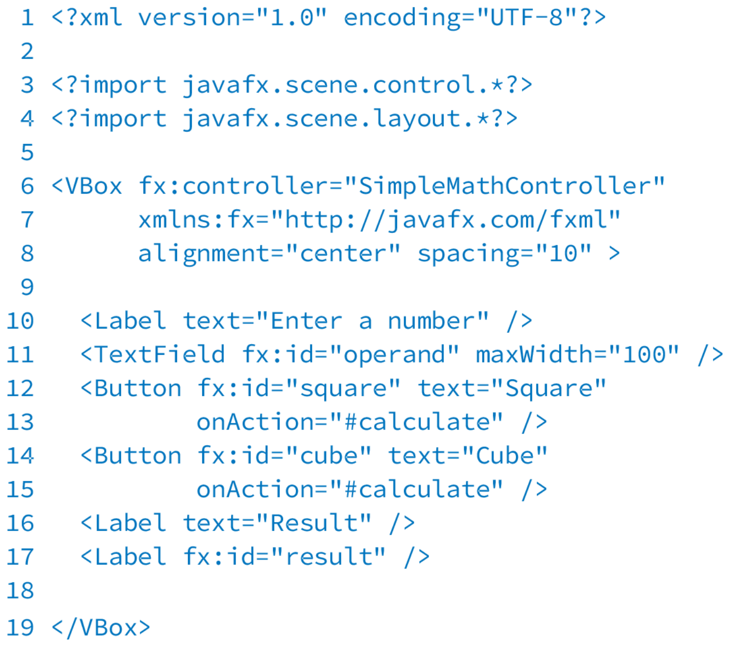
Example 17.5 The Model, *SimpleMath.java*

```
/*  Simple Math Class
    Anderson, Franceschi
*/

public class SimpleMath
{
  /** square method
  *  @param operand  number to square
  *  @return the square of operand
  */
  public static double square( double operand )
  {
    return operand * operand;
  }

  /** cube method
  *  @param operand  number to cube
  *  @return the cube of operand
  */
  public static double cube( double operand )
  {
    return operand * operand * operand;
  }
}
```

Description

The controls and their layout are defined in the FXML file, shown in **Example 17.6**, which acts as our View. Because we want to respond to the user clicking either button, we define a controller at line 6 using the *fx:controller* attribute of the layout container with its value set to the class name of the Java file that contains our controller. Thus, in this case, the controller is *SimpleMathController.java.*

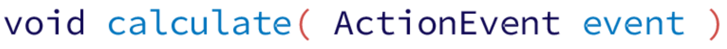
Example 17.6 The View, *fxml_simple_math.fxml*



```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import javafx.scene.control.*?>
4 <?import javafx.scene.layout.*?>
5
6 <VBox fx:controller="SimpleMathController"
7       xmlns:fx="http://javafx.com/fxml"
8       alignment="center" spacing="10" >
9
10   <Label text="Enter a number" />
11   <TextField fx:id="operand" maxWidth="100" />
12   <Button fx:id="square" text="Square"
13           onAction="#calculate" />
14   <Button fx:id="cube" text="Cube"
15           onAction="#calculate" />
16   <Label text="Result" />
17   <Label fx:id="result" />
18
19 </VBox>
```

Description

The two buttons are defined at lines 12 through 15. The *text* attribute specifies the wording to appear on the button. The *onAction* attribute specifies a method in the controller that should be executed when the user clicks the button. Thus, our controller must define a method named *calculate*. Usually, we code the method as accepting an *ActionEvent* object, as shown below, although this is not required.



```
void calculate( ActionEvent event )
```

Description

The *ActionEvent* object is automatically created when the user clicks a button, selects an item from a list or a menu, or presses the *Enter* key in a *TextField*. The *ActionEvent* object contains data we can query, such as which control fired the event.

The *calculate* method is our event handler or listener, and by setting the *onAction* attribute for a control, we **register** the event handler on our control: our buttons, in this example. Thus, when the user clicks on either of the buttons, the *calculate* method executes.

We have also used the *fx:id* attribute to define a name for all the controls that the controller needs to access. Assigning an *fx:id* to a control puts the control in the *fx* namespace, making the control accessible outside the file in which the control is defined. This allows the controller to access controls using their *fx:id* value.

Our controller is shown in **Example 17.7**. It defines as instance variables all the controls it needs to access using their *fx:id* names that we defined in the FXML file (lines 7–10). We use the *@FXML* annotation in front of the instance variable definitions to indicate that these controls, although defined as *private*, are also accessible to FXML.

Example 17.7 The Controller, *SimpleMathController.java*

```
 1 import javafx.event.ActionEvent;
 2 import javafx.fxml.FXML;
 3 import javafx.scene.control.*;
 4
 5 public class SimpleMathController
 6 {
 7   @FXML private TextField operand;
 8   @FXML private Label result;
 9   @FXML private Button square;
10   @FXML private Button cube;
11
12   @FXML protected void calculate( ActionEvent event )
13   {
14     try
15     {
16       double op = Double.parseDouble( operand.getText( ) );
17       if ( event.getSource( ) == square )
18         result.setText( String.valueOf( SimpleMath.square( op ) ) );
19       else if ( event.getSource( ) == cube )
20         result.setText( String.valueOf( SimpleMath.cube( op ) ) );
21     }
22     catch ( NumberFormatException nfe )
23     {
24       operand.setText( "" );
25       result.setText( "???" );
26     }
27   }
28 }
```

Description

The *calculate* method is defined at lines 12–27. We also precede the method header with the *@FXML* annotation, again to give FXML access to this *protected* method. The *calculate* method takes as a parameter an *ActionEvent* object. As mentioned, this object is created when the user clicks on the button and contains information about the event that occurred.

At line 16, we use the *getText* method to extract the text that the user typed into the *operand TextField.* The *getText* method is shown along with its companion *setText* method in **Table 17.8**. Because the *getText* method returns a *String,* we call the *static parseDouble* method of the *Double* wrapper class to convert the *String* to a *double*. We enclose this operation in a *try/catch* block (lines 14–26) because the *parseDouble* method throws a *NumberFormatException* if the text the user entered cannot be converted to a *double*.

Useful Methods of the *TextField* Class

Package	**javafx.scene.control**
Return value	**Method name and argument list**
`String`	`getText( )`
returns the text typed into the *TextField*	
`void`	`setText( String newText )`
sets the text in the *TextField* to *newText*	

Description

If the conversion goes well, we then determine which button the user pressed. The *ActionEvent* class provides a useful method, *getSource*, shown in **Table 17.9**, which we can use to determine which button the user actually clicked. At line 17, we test whether the user clicked the *square* button. If so, we call the *static square* method in our *SimpleMath* Model, passing the converted number and setting the *result Label* to the return value. Because the *setText* method requires a *String* argument, we call the *static valueOf* method of the *String* class to convert the return value to a *String*.

The *getSource* Method of the *ActionEvent* Class

Package	**javafx.event**
Return value	**Method name and argument list**
`Object`	`getSource( )`
returns the object on which the event was triggered	

Description

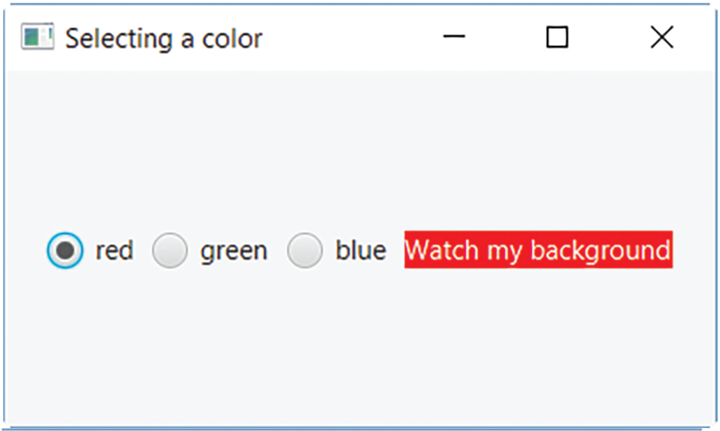
Similarly, at lines 19 and 20, we test whether the user clicked the *cube Button*; if so, we call the *static cube* method of the *SimpleMath* class and set the *result Label* to the return value.

Finally, *SimpleMathPractice*, the launch class, shown in **Example 17.8**, performs the usual operations of loading the *fxml_simple_math.fxml* file, setting the scene and stage parameters, and displaying the prepared window.

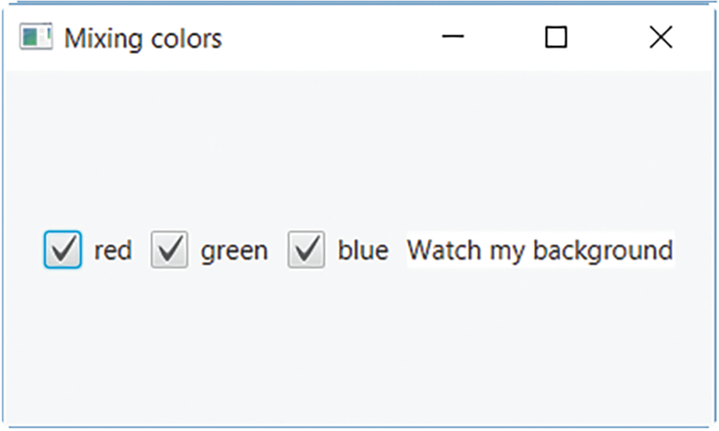
Example 17.8 *SimpleMathPractice.java*

```java
/*  Simple Math Operations Using Buttons
    Anderson, Franceschi
*/

import java.net.URL;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.layout.VBox;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class SimpleMathPractice extends Application
{
  @Override
  public void start( Stage stage ) // throws Exception
  {
    try
    {
      URL url =
         getClass( ).getResource( "fxml_simple_math.fxml" );
      VBox root = FXMLLoader.load( url );
      Scene scene = new Scene( root, 300, 275 );
      stage.setTitle( "Simple Math" );
      stage.setScene( scene );
      stage.show( );
    }
    catch ( Exception e )
    {
      e.printStackTrace( );
    }
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description



● red ○ green ○ blue Watch my background



☑ red ☑ green ☑ blue Watch my background

17.6 Radio Buttons and Checkboxes

If you have ever completed a survey on the web, you are probably familiar with radio buttons and checkboxes.

Radio buttons prompt the user to select one of several mutually exclusive options. Clicking on any radio button deselects any previously selected radio button. Thus, in a group of radio buttons, a user can select only one option at a time.

Checkboxes are often associated with the instruction “check all that apply”; that is, the user is asked to select 0, 1, or more options. A checkbox is a toggle button in that if the option is not currently selected, clicking on a checkbox selects the option; if the option is currently selected, clicking on the checkbox deselects the option.

We present two similar examples to illustrate how to use the *RadioButton* and *CheckBox* classes and how they differ. Both examples allow the user to select the background color for a label. We display three color options: red, green, and blue. Using radio buttons, only one color can be selected at a time. Thus, by clicking on a radio button, the user causes the background of the label to be displayed in one of three colors. Using checkboxes, the user can select any combination of the three color options, so the label color can be set to any of eight possible combinations.

We start with the example using radio buttons. The window when the application starts running is shown in **Figure 17.6**.

Figure 17.6 The GUI for the *ChangingColors* Application

Description

We begin with the Model, *ColorSelector.java*, shown in **Example 17.9**. Again, this model is minimal, having only one *static* method, *colorToHexString* (lines 11–25). For convenience in passing parameters, we define three *public static* constants at lines 7 through 9. The controller uses these constants as arguments for the *colorToHexString* method.

Example 17.9 The Model, *ColorSelector.java*

```
/*  ColorSelector class
    Anderson, Franceschi
*/

public class ColorSelector
{
  public static final int RED = 0;
  public static final int GREEN = 1;
  public static final int BLUE = 2;

  /** colorToHexString method
  *  @param selection  the selected color
  *  @return the hex representation of the selec
  */
  public static String colorToHexString( int sel
  {
    String result = "#";
    if ( selection == RED )
      result += "FF0000";
    else if ( selection == GREEN )
      result += "00FF00";
    else if ( selection == BLUE )
      result += "0000FF";
    return result;
  }
}
```

Description

Next, we look at the View, the FXML file that defines the layout and the controls for the GUI, which is shown in **Example 17.10**. For this application, we use an *HBox* layout (lines 6–28), which arranges controls horizontally. We set attributes (line 8) to specify that the controls should be centered with 10 pixels between each control. We also specify the class of the controller that handles the events caused by the user selecting a radio button, as *ChangingColorsController* (line 6).

In order for the radio buttons to be mutually exclusive (i.e., selecting one radio button deselects any previously selected radio button), we define a *ToggleGroup* (lines 10–12). We use the *<fx:define>* element to create objects, such as this *ToggleGroup*, that are not in the scene graph but need to be referenced by a control. In the definition of each radio button (lines 14–22), we add the radio button to the group by setting its *toggleGroup* property to the id of the *ToggleGroup*. We preface the id with "$" to indicate that *colorGroup* is a variable, rather than a predefined property value.

For each radio button, we also use the *onAction* property to specify that the *colorChosen* method in the *ChangingColorsController* should be executed when the user clicks the radio button.

At line 14, we set the red radio button's *selected* property to *true*. This automatically sets the radio button as selected when the program starts.

REFERENCE POINT

To learn more about using CSS for styling controls, read the JavaFX CSS Reference Guide at *https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html*.

Finally, we define the label whose background we color as the user clicks on the radio buttons (lines 24–26). Here we introduce *style* definitions. If you are familiar with Cascading Style Sheets (CSS) used with HTML pages, this should look familiar to you. We explain how to use CSS at the end of this chapter. The *background-color* property sets the background color of a control. Its value can be either a CSS named constant (such as *red*) or the RGB hexadecimal value of the color preceded by a "#" (such as *#FF0000*).

We set the label's background to the color red (*#FF0000*) to be consistent with the *red* radio button being selected when the application starts.

Example 17.10 *fxml_changing_colors.fxml*

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<HBox fx:controller="ChangingColorsController"
      xmlns:fx="http://javafx.com/fxml"
      alignment="center" spacing="10" >

  <fx:define>
    <ToggleGroup fx:id="colorGroup" />
  </fx:define>

  <RadioButton fx:id="red" text="red" selected="true"
               toggleGroup="$colorGroup"
               onAction="#colorChosen" />
  <RadioButton fx:id="green" text="green"
               toggleGroup="$colorGroup"
               onAction="#colorChosen" />
  <RadioButton fx:id="blue" text="blue"
               toggleGroup="$colorGroup"
               onAction="#colorChosen" />

  <Label fx:id="label" text="Watch my background"
         textFill="WHITE"
         style="-fx-background-color:#FF0000" />

</HBox>
```

Description

Table 17.10 summarizes the special FXML language elements that we have used.

FXML Language Elements and Their Meaning

FXML Element	**Meaning**
fx:define	Used to create objects not in the scene graph that need to be referenced later
FXML Attributes	**Meaning**
fx:id	Defines a name that can be referenced across the FXML application

fx:controller	Defines a class that contains event handling code for one or more controls
onAction	Defines a method name (preceded by “#”) in the controller that should be executed when the user interacts with the control
FXML Annotation	**Meaning**
@FXML	Allows FXML to access a *private* or *protected* class, method, or data
FXML Prefixes	**Meaning**
-fx-	Used to distinguish a JavaFX style attribute from a CSS attribute
$	Used as a prefix to a variable name when the variable is used as a property value
@	Used as a prefix for a URI to specify that the path of the URI starts with the current folder

Description

The Controller, *ChangingColorsController.java*, is shown in **Example 17.11**. At lines 11 through 14, we define instance variables for all the controls, using the *fx:id* values we defined in the FXML file. The *colorChosen* event handler method, defined at lines 16 through 31, takes as a parameter the *ActionEvent* that is generated when the user clicks on a radio button. We want to set the background color of the label to correspond with the radio button that was clicked. Thus, we set the *style* property of the label. We start by setting a *String* to characters that are common for any color selected (line 18). We then use the *getSource* method of the *ActionEvent* to determine which radio button was clicked and call the *colorToHexString* method in our Model, passing as a parameter one of the *public static* constants (*RED*, *GREEN*, or *BLUE*) defined in the Model. We then append the returned hexadecimal *String* to *style*. At line 30, we call the *setStyle* method of the *Label* class, passing it the completed *String*. Note that for any property that can be set using attributes in the FXML file, JavaFX usually also provides accessor and mutator methods in the GUI control’s class for setting or getting the property programmatically.

Example 17.11 The Controller, *ChangingColorsController.java*

```
/*  ChangingColorsController class
    Anderson, Franceschi
*/

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.*;

public class ChangingColorsController
{
  @FXML private RadioButton red;
  @FXML private RadioButton green;
  @FXML private RadioButton blue;
  @FXML private Label label;

  @FXML protected void colorChosen( ActionEvent event )
  {
    String style = "-fx-background-color: ";

    if ( event.getSource( ) == red )
     style +=
      ColorSelector.colorToHexString( ColorSelector.RED );
    else if ( event.getSource( ) == green )
     style +=
      ColorSelector.colorToHexString( ColorSelector.GREEN );
    else if ( event.getSource( ) == blue )
     style +=
      ColorSelector.colorToHexString( ColorSelector.BLUE );

    label.setStyle( style );
  }
}
```

Description

SOFTWARE ENGINEERING TIP

Ensure that the initial state of the GUI controls represents the state of the Model.

The last part of the application is *ChangingColors*, the launching class, shown in **Example 17.12**. It follows the format of the previous application launching classes.

Example 17.12 The Application Launcher: *ChangingColors.java*

```
/*  Select a Color Using RadioButtons
    Anderson, Franceschi
*/

import java.net.URL;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.layout.HBox;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class ChangingColors extends Application
{
  @Override
  public void start( Stage stage )
  {
    try
    {
      URL url =
       getClass( ).getResource( "fxml_changing_colors.fxml" );
      HBox root = FXMLLoader.load( url );
      Scene scene = new Scene( root, 400, 200 );
      stage.setTitle( "Selecting a color" );
      stage.setScene( scene );
      stage.show( );
    }
    catch ( Exception e )
    {
      e.printStackTrace( );
    }
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description

We now demonstrate a similar application using *CheckBoxes*. For this example, we create the interface shown in **Figure 17.7**.

Figure 17.7 The GUI for the *MixingColors* Application

Description

Example 17.13 shows the Model, *ColorMixer.java.* The Model's job is to manage the color mixing based on which colors are selected or deselected. As in our *RadioButton* example, the *ColorMixer* also provides *public static* constants (lines 7–9) for the controller's convenience in passing parameters. To manage which colors are selected, the *ColorMixer* defines an array of three *boolean* values (line 11), with each element representing whether the corresponding color is selected *(true)* or deselected (*false*).

Example 17.13 The Model, *ColorMixer.java*

```java
/*  ColorMixer class
    Anderson, Franceschi
*/

public class ColorMixer
{
  public static final int RED = 0;
  public static final int GREEN = 1;
  public static final int BLUE = 2;

  private boolean [ ] rgb;

  /** default constructor
  *   sets all elements in the rgb array to true
  *   to represent the color white
  */
  public ColorMixer( )
  {
    rgb = new boolean[3];
    rgb[RED] = true;
    rgb[GREEN] = true;
    rgb[BLUE] = true;
  }

  /** toggleColor method
  *   toggles the color on/off
  *   @param color  the color to be toggled
  */
  public void toggleColor( int color )
  {
    rgb[color] = !rgb[color];
  }

  /** hexStringColor
  *  @return the hexadecimal representation
  *          of the color mix
  */
  public String hexStringColor( )
  {
    String result = "#";

    result += ( rgb[RED] ? "FF" : "00" );
    result += ( rgb[GREEN] ? "FF" : "00" );
    result += ( rgb[BLUE] ? "FF" : "00" );

    return result;
  }
}
```

Description

The constructor (lines 13–23) instantiates the array and sets all elements to *true*, representing the color white (all colors selected). The *RED*, *GREEN*, and *BLUE* constants also come in handy here as logical names for the array indexes.

The *toggleColor* method (lines 25–32) alternates the state of a color between *true* and *false*. This is consistent with the behavior of *CheckBoxes,* where each click of the *CheckBox* changes its state between selected and deselected.

Finally, the *hexStringColor* method (lines 34–47) composes a *String* containing the hexadecimal equivalent of the current color. Using the conditional operator, we set each color's contribution to the *String* as either *"FF"* (full color) if selected or *"00"* (no color) if deselected.

Next, the FXML file, *fxml_mixing_colors.fxml,* shown in **Example 17.14**, defines the GUI. At lines 11 through 16, we define the three *CheckBoxes* to represent the red, green, and blue color components. We also preselect each *CheckBox* and set the *Label* background to white (line 19) to be consistent with the original state of the Model.

Example 17.14 *fxml_mixing_colors.fxml*

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2
 3 <?import javafx.scene.control.*?>
 4 <?import javafx.scene.layout.*?>
 5
 6 <HBox fx:controller="MixingColorsController"
 7       xmlns:fx="http://javafx.com/fxml"
 8       alignment="center" spacing="10"
 9       style="-fx-background-color:#CCCCCC" >
10
11   <CheckBox fx:id="red" selected="true"
12             text="red" onAction="#mix" />
13   <CheckBox fx:id="green" selected="true"
14             text="green" onAction="#mix" />
15   <CheckBox fx:id="blue" selected="true"
16             text="blue" onAction="#mix" />
17
18   <Label fx:id="label" text="Watch my background"
19          style="-fx-background-color:#FFFFFF" />
20
21 </HBox>
```

Description

Our controller for this application, *MixingColorsController*, is named at line 6, and we specify that the *mix* method in the controller is the handler for a user selecting or deselecting any of the *CheckBoxes* (lines 12, 14, and 16).

The controller, *MixingColorsController.java,* is shown in **Example 17.15**. The methods in this Model are not *static*, so we define an instance variable, *mixer*, representing the Model (line 11) and the controller instantiates the Model in the constructor (lines 18–21). The *mix* event handler method (lines 23–35) calls the *getSource* method of the *ActionEvent* object to determine which *CheckBox* was clicked. It calls the *toggleColor* method in the Model to invert the state of the color in the *rgb* array. In this way, we do not need to check whether the *CheckBox* was just selected or deselected. Once the clicked color component is updated, we compose the new color *String* by calling the *hexStringColor* method of the model (line 33); then we use that *String* to set the *label*'s background color.

Example 17.15 The Controller, *MixingColorsController.java*

```
/*  MixingColorsController class
    Anderson, Franceschi
*/

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.*;

public class MixingColorsController
{
  private ColorMixer mixer;

  @FXML private CheckBox red;
  @FXML private CheckBox green;
  @FXML private CheckBox blue;
  @FXML private Label label;

  public MixingColorsController( )
  {
    mixer = new ColorMixer( );
  }

  @FXML protected void mix( ActionEvent event )
  {
    if ( event.getSource( ) == red )
      mixer.toggleColor( ColorMixer.RED );
    else if ( event.getSource( ) == green )
      mixer.toggleColor( ColorMixer.GREEN );
    else if ( event.getSource( ) == blue )
      mixer.toggleColor( ColorMixer.BLUE );

    String style = "-fx-background-color: ";
    style += mixer.hexStringColor( );
    label.setStyle( style );
  }
}
```

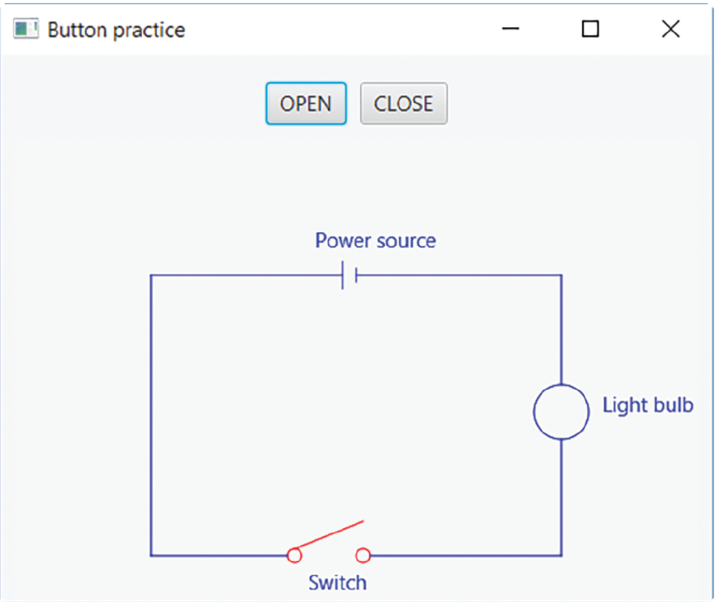
Description

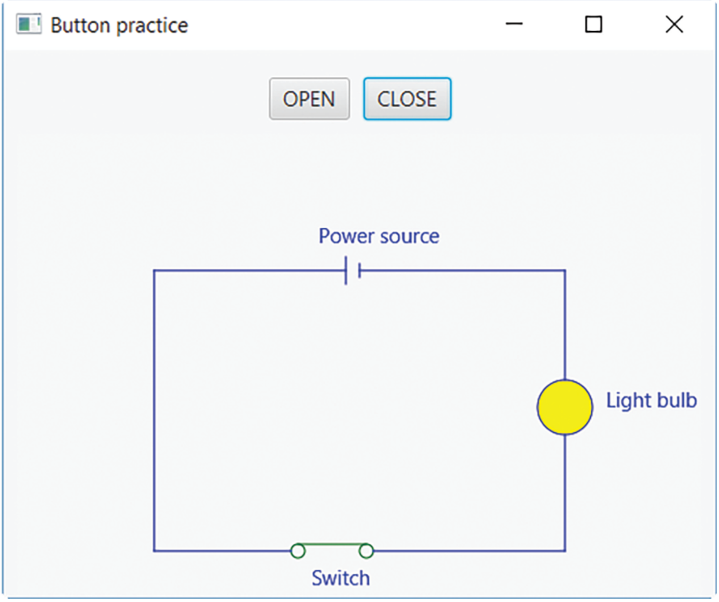
The final piece to this application is the standard launch class, shown in **Example 17.16**.

Example 17.16 The Launch Class, *MixingColors.java*

```java
/*  Mixing Colors Using CheckBoxes
    Anderson, Franceschi
*/

import java.net.URL;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.layout.HBox;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class MixingColors extends Application
{
  @Override
  public void start( Stage stage )
  {
    try
    {
      URL url =
         getClass( ).getResource( "fxml_mixing_colors.fxml" );
      HBox root = FXMLLoader.load( url );
      Scene scene = new Scene( root, 400, 200 );
      stage.setTitle( "Mixing colors" );
      stage.setScene( scene );
      stage.show( );
    }
    catch ( Exception e )
    {
      e.printStackTrace( );
    }
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description


Button practice
OPEN
CLOSE
Power source
Light bulb
Switch


Button practice
OPEN
CLOSE
Power source
Light bulb
Switch

17.7 Programming Activity 1: Working with Buttons

In this activity, you will work with two *Buttons* that control a simulated electrical switch. Specifically, you will write the code to perform the following operations:

1. If the user clicks on the “OPEN” button, open the switch.
2. If the user clicks on the “CLOSE” button, close the switch.

The framework for this Programming Activity will animate your code so that you can check its accuracy. **Figures 17.8** and **17.9** show the application after the user has clicked the button labeled “OPEN” and the button labeled “CLOSE,” respectively.

Figure 17.8 User Clicked “OPEN”

Description

Figure 17.9 User Clicked “CLOSE”

Description

Instructions

Copy the source files in the Programming Activity 1 folder for this chapter to a folder on your computer. Note that all files should be in the same folder.

Open the *fxml_button_practice.fxml* file. Searching for five asterisks (*****) in the source code will position you to the first code section and then to the second location where you will add your code. In task 1, you will specify the name of the controller as *ButtonPracticeController*. In task 2, you will define an *HBox* layout with the two buttons. **Example 17.17** shows the section of the *fxml_button_practice.fxml* file where you will add your code.

Example 17.17 Location of Student Code in *fxml_button_practice.fxml*

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<!-- ***** 1. Student code starts here -->
<!-- add the code to this element definition to specify
     the controller as ButtonPracticeController -->
<BorderPane fx:id="bp"
            xmlns:fx="http://javafx.com/fxml" >

  <top>
    <!-- ***** 2. Student code restarts here -->
    <!-- add code to define an HBox layout that is centered
         with an id of hBox and 10 pixels between controls.
         Inside the HBox, add 2 Buttons, with text OPEN and
         CLOSE, and ids of open and close. Clicking on either
         button triggers a call to the event handler flip
         method -->



  </top>

</BorderPane>
```

Description

Note that we define an *HBox* layout container inside a *BorderPane* layout container, illustrating that layouts can be nested. We demonstrate nested layouts again later in the chapter.

Next, open *ButtonPracticeController.java* and search for five asterisks (*****). This will position you at the third and final task for this Programming Activity, writing the event handler for the buttons. **Example 17.18** shows the section of the source code where you will add your code.

Example 17.18 Location of Student Code in *ButtonPracticeController.java*

```
// ***** 3. Student code restarts here
// Code the flip method.
// To open the switch, call the open method
//   with circuit, the object reference of the
//   Circuit object.
//   The open method does not take any parameters.
// To close the switch, call the close method
//   with circuit.
//   The close method does not take any parameters.
// The last statement of the method should be
//   animate( );
```

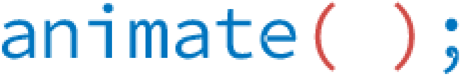
Description

Our framework will animate your code so that you can watch your code work. For this to happen, be sure that you call the *animate* method as the last statement in the *flip* method.

Troubleshooting

If your *flip* method does not animate, check these tips:

- Verify that the last statement in your *flip* method is:

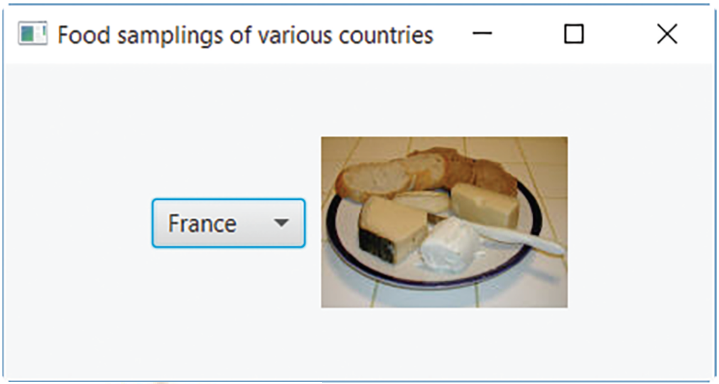


```
animate( );
```

- Verify that your listener is registered on the buttons (*onAction*).
- Verify that you have correctly identified the button that fired the event using the *getSource* method.

DISCUSSION QUESTIONS

1. Explain why the *getSource* method is useful here.
2. Could you implement this application with *RadioButtons* instead of *Buttons*? What definition would you need to add to the *FXML* file? How would the *flip* method change, if at all?
3. Which class is the Model?



17.8 Combo Boxes

A *ComboBox* implements a drop-down list. When the combo box appears, either no item or one item is displayed, along with a down arrow icon. When the user presses on the down arrow, the combo box “drops” open and displays a list of items, with a scroll bar for viewing more items. The user can select one item from the list. When the user selects an item, the list closes and only the selected item is displayed.

In this example, we allow the user to select a country from a combo box, and in response, we display an image of a typical food from the selected country. **Figure 17.10** shows the window when our application begins.

Figure 17.10 The Food Sampling Application

Description

We begin as usual with the Model, *FoodSampler.java*, shown in **Example 17.19**. Lines 9 through 16 define two arrays. The *countryList* array contains the country names to be displayed in the *ComboBox*, and the *foods* array contains the corresponding images of food stored in the same order as the country names. The instance variable *selectedInde*x, defined at line 17, holds the index of the selected country. We use this index to retrieve the appropriate food image to display. With s*electedIndex* set to 0, the Model's initial state is that the country name "France" is selected and *cheese.jpg* is the corresponding image.

Example 17.19 The Model, *FoodSampler.java*

```java
/*  FoodSampler class
    Anderson, Franceschi
*/

import javafx.scene.image.Image;

public class FoodSampler
{
  private String [ ] countryList =
     { "France", "Greece", "Italy", "Japan", "USA" };
  private Image [ ] foods =
     { new Image( "cheese.jpg" ),
       new Image( "fetasalad.jpg" ),
       new Image( "pizza.jpg" ),
       new Image( "sushi.jpg" ),
       new Image( "hamburger.jpg" ) };
  private int selectedIndex = 0;

  /** getCountryList method
  *  @return a reference to the countryList array
  */
  public String [ ] getCountryList( )
  {
    return countryList;
  }

  /** the accessor for selectedIndex
  *  @return the index selected
  */
  public int getSelectedIndex( )
  {
    return selectedIndex;
  }

  /** the mutator for selectedIndex
  * @param selection the new value for selectedIndex
  */
  public void updateSelection( int selection )
  {
    selectedIndex = selection;
  }

  /** the accessor for the image to display
  *  @return the image from the Image array
  *          using the index selectedIndex
  */
  public Image getImageSelected( )
  {
    return foods[selectedIndex];
  }
}
```

Description

The remainder of the Model consists of accessor methods for the *countryList* array (lines 19–25) and *selectedIndex* (lines 27–33), a mutator method for *selectedIndex* (lines 35–41), and an accessor for the image corresponding to *selectedIndex* (lines 43–50).

Next we look at the FXML file, *fxml_food_samplings.fxml,* shown in **Example 17.20**. At line 7, we identify the Controller as *FoodSamplingsController.java*. Lines 11 through 13 define our *ComboBox* control and that the *itemSelected* method in the Controller class will handle the event fired by a user selecting an item from the *ComboBox* drop-down list. We also set the *visibleRowCount* property to 3, specifying that when the user opens the *ComboBox*, three items should be displayed along with a scrollbar for viewing the other hidden items. The default value for the *visibleRowCount* property is 10. If the list contains fewer items than specified in the *visibleRowCount* property, the *ComboBox* displays all the items in the list.

Example 17.20 The *fxml_food_samplings.fxml File*

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2
 3 <?import javafx.scene.control.*?>
 4 <?import javafx.scene.image.*?>
 5 <?import javafx.scene.layout.*?>
 6
 7 <HBox fx:controller="FoodSamplingsController"
 8       xmlns:fx="http://javafx.com/fxml"
 9       alignment="center" spacing="10" >
10
11   <ComboBox fx:id="countries" visibleRowCount="3"
12             onAction="#itemSelected" >
13   </ComboBox>
14
15   <ImageView fx:id="foodImage">
16   </ImageView>
17
18 </HBox>
```

Description

At lines 15 and 16 we also define an *ImageView* control to display the appropriate food image. We do not specify the initial image here; we let the Controller get the appropriate image from the Model.

We turn now to the Controller, *FoodSamplingsController.java*, shown in **Example 17.21**. New to this Controller is the *initialize* method, which is called after the scene graph has been created, and can be used to add items to the scene graph that could not be fully defined in the FXML file. Because the Model stores the list of countries and corresponding images, we cannot specify the items for *ComboBox* in the FXML file. However, this approach also has some advantages: By adding the items to the *ComboBox* at runtime, we can easily handle changes in the list without needing to change our Controller or FXML code, and we can use this application with another Model that has a different set of items and images.

Example 17.21 The Controller, *FoodSamplingsController.java*

```
 1 /*  FoodSamplingsController class
 2     Anderson, Franceschi
 3 */
 4
 5 import javafx.event.ActionEvent;
 6 import javafx.fxml.FXML;
 7 import javafx.scene.control.*;
 8 import javafx.scene.image.*;
 9
10 public class FoodSamplingsController
11 {
12   private FoodSampler sampler;
13
14   @FXML private ComboBox<String> countries;
15   @FXML private ImageView foodImage;
16   private SingleSelectionModel<String> selectionModel;
17
18   public void initialize( )
19   {
20    sampler = new FoodSampler( );
21
22    // populate combobox with data from the Model
23    countries.getItems( ).addAll( sampler.getCountryList( ) );
24
25    // get a reference to the SingleSelectionModel
26    selectionModel = countries.getSelectionModel( );
27
28    // initialize View with initial data from Model
29    selectionModel.select( sampler.getSelectedIndex( ) );
30    foodImage.setImage( sampler.getImageSelected( ) );
31   }
32
33   @FXML protected void itemSelected( ActionEvent event )
34   {
35    // retrieve index of country selected
36    int index = selectionModel.getSelectedIndex( );
37    // update the Model
38    sampler.updateSelection( index );
39    // update the View with Image from the Model
40    foodImage.setImage( sampler.getImageSelected( ) );
41   }
42 }
```

Description

A *ComboBox* control is a generic class, so when the Controller defines the *ComboBox* (line 14), we need to specify the class type of the items to be displayed; in this example, we are displaying *Strings*.

A *ComboBox*, like other controls that allow users to select items, uses a *SelectionModel* to handle the user's interactions with the control. The *ComboBox* uses the

SingleSelectionModel, which is a subclass of *SelectionModel*, to restrict the user to a choice of only one item from the list. The *SingleSelectionModel* class is also a generic class; thus, we specify the class type to be *String* in line 16. **Table 17.11** shows some useful methods of the *SingleSelectionModel* class for setting and getting the user's selection.

Useful Methods of the *SingleSelectionModel* Class

Package	**javafx.scene.control**
Return value	**Method name and argument list**
`int`	`getSelectedIndex( )`
returns the index of the currently selected item.	
`T`	`getSelectedItem( )`
returns the currently selected item.	
`void`	`select( int index )`
selects the item at *index*. Any previously selected item is deselected.	

Description

In order for our Controller event handler method, *itemSelected,* to be notified when the user selects an item, we need to put our items into an *ObservableList* object. **Table 17.12** shows the *getItems* method of the *ComboBox*, which returns an *ObservableList*. The *addAll* method of the O*bservableList* class, shown in **Table 17.13**, can be used to insert items into the list. The *addAll* method uses the *varargs* syntax to accept a variable number of arguments.

A Useful Method of the *ComboBox<T>* Class

Package	**javafx.scene.control**
Return value	**Method name and argument**

	list
`ObservableList`	`getItems( )`
returns the values of the items in the *ComboBox* as an *ObservableList*	

Description

A Useful Method of the *ObservableList<E>* Interface

Package	**javafx.collections**
Return value	**Method name and argument list**
`boolean`	`addAll( E... elements )`
adds the elements to the list	

Description

Putting this all together, in the *initialize* method, we instantiate the Model (line 20), and then call the *getItems* method, which returns an empty *ObservableList.* Next, we call the *addAll* method of the *ObservableList* class to insert into the *ComboBox* the array of country names returned by the Model's *getCountryList* method (lines 22–23).

At line 26, we get a reference to the *SingleSelectionModel*; we then use that reference to select the item using the index returned from the Model's *getSelectedIndex* method (line 29). At this point, the value returned will be 0, because in the Model, we initialize the *selectedIndex* instance variable to 0. We then set the image for our *foodImage ImageView* to the filename returned by the Model's *getImageSelected* method (line 30). Thus, when the window first appears, the country *France* is selected and we display the *cheese.jpg* image.

The *itemSelected* event handler, which executes whenever the user selects a new item from the *ComboBox* (lines 33–41), retrieves the index of the selected item by calling the *getSelectedIndex* method, using the *selectionModel* reference. Using that index, we notify the model that the user has changed the selection. Then we determine which new image to display by calling the Model's *getImageSelected* method.

The last piece of the application is the launch class, shown in **Example 17.22**.

Example 17.22 *FoodSamplings.java*

```java
/*  Using ComboBox to show a sampling of international foods
    Anderson, Franceschi
*/

import java.net.URL;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.layout.HBox;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class FoodSamplings extends Application
{
  @Override
  public void start( Stage stage )
  {
   try
   {
     URL url =
        getClass( ).getResource( "fxml_food_samplings.fxml" );
     HBox root = FXMLLoader.load( url );
     Scene scene = new Scene( root, 450, 200 );
     stage.setTitle( "Food samplings of various countries" );
     stage.setScene( scene );
     stage.show( );
   }
   catch ( Exception e )
   {
     e.printStackTrace( );
   }
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

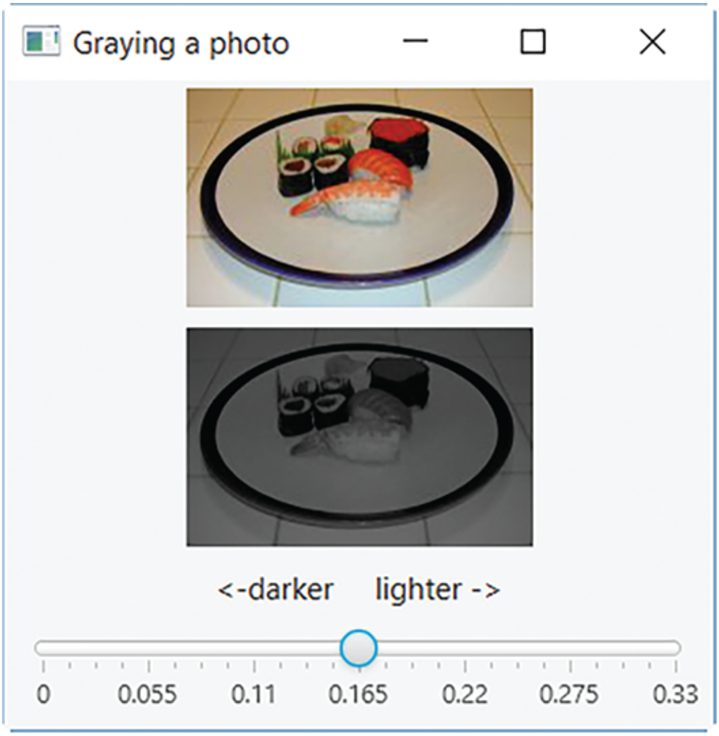
Description

CODE IN ACTION

Within the online resources, you will find a movie with a

step-by-step illustration of working with GUI components. Click on the link to start the movie.

© Hemera Technologies/Photos.com/Thinkstock



17.9 Sliders

The *Slider* control is capable of displaying a set of continuous values along a horizontal or vertical line, called a **track**. The user "slides" the knob, called the **thumb**, along the track to select a value from the set. To illustrate the *Slider* control, we create an application that allows the user to control the level of gray in a photo. A gray color is created when the red, green, and blue components have the same value. Each color component's value can range from 0.0 to 1.0.

Figure 17.11 shows the GUI when the application begins. The top photo is the original version of the image; this may look familiar as the *sushi.jpg* photo from the *ComboBox* example in the last section. The lower photo has been converted to a middle level of gray. As the user slides the knob left, we darken the photo; as the user slides the knob to the right, we lighten the photo. The vertical lines and numbers shown under the slider are **tick marks** and **tick values.** The values represent a multiplier we use to convert the red, green, and blue color values of the photo to a new level of gray.

Figure 17.11 The Photo Graying GUI

Description

We define the *Slider* control in the FXML file, shown in **Example 17.23.** At lines 20 through 23, we specify the minimum multiplier value as 0, the maximum as .33, and the initial value in the middle of that range as .165. We set properties to show the tick marks and tick labels and to display the tick marks at intervals of .055 units.

Example 17.23 The View, *fxml_photo_grayer.fxml*

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2
 3 <?import javafx.scene.control.*?>
 4 <?import javafx.scene.image.*?>
 5 <?import javafx.scene.layout.*?>
 6
 7 <VBox fx:controller="PhotoGrayerController"
 8     xmlns:fx="http://javafx.com/fxml"
 9     alignment="center" spacing="10"
10     style="-fx-padding:0 10 0 10;" >
11
12   <ImageView fx:id="originalImageView" >
13     <Image url = "Sushi.jpg" />
14   </ImageView>
15
16   <ImageView fx:id="grayImageView" />
17
18   <Label text="&lt;-darker lighter -&gt;" />
19
20   <Slider fx:id="slider" min="0" max=".33"
21           value=".165" showTickMarks="true"
22           showTickLabels="true" majorTickUnit="0.055" />
23
24 </VBox>
```

Description

COMMON ERROR TRAP

Be sure to use the XML and FXML encoding sequences when specifying text values that contain characters having syntactic meaning in XML and FXML. Otherwise, the FXML Loader will generate an error.

In line 10, we add a *style* property to define padding values for the *VBox*. We add 10 pixels of padding on the left and right sides of the *VBox,* but no padding on the top and bottom.

SOFTWARE ENGINEERING TIP

Labels can help guide the user through the interface.

The *Label* control defined at line 18 aids the user by showing that moving the thumb to the left will darken the image and moving the thumb to the right will lighten the image. Because angle brackets have syntactic meaning in FXML, we use the special character encoding sequences *&lt;* and *&gt;* to represent the left and right angle brackets, respectively. **Table 17.14** shows the encoding sequences for the XML and FXML special characters. The five XML encoding sequences start with an ampersand and end with a semicolon. In addition, FXML adds three special characters; the encoding sequences for these FXML special characters consist simply of a leading backslash followed by the special character. These encoding sequences are useful whenever a control's text value contains any of these special characters as data.

XML and FXML Special Characters

Character	**XML Encoding Sequence**
<	&lt;
>	&gt;
&	&amp;
"	&quot;
'	&apos;
Character	**FXML Encoding Sequence**
@	\@
$	\$
%	\%

Description

By looking at the Model (**Example 17.24**), we can see how the *Slider's* multiplier values are used to adjust the image's color. The Model, *PhotoGrayer.java,* extends the *Image* class, which is used to load graphical images (GIF, BMP, JPG, and PNG) from a URL. In our constructor (lines 13–21), we pass the image filename to the constructor of the *Image* superclass, shown in **Table 17.15**.

Example 17.24 The Model, *PhotoGrayer.java*

```java
/*  Adjusts the gray value of an image
    Anderson, Franceschi
*/

import javafx.scene.image.Image;
import javafx.scene.image.PixelReader;
import javafx.scene.image.PixelWriter;
import javafx.scene.image.WritableImage;
import javafx.scene.paint.Color;

public class PhotoGrayer extends Image
{
  /** constructor
  * @param file  filename of the image
  *    passes the filename to the Image class
  *    constructor
  */
  public PhotoGrayer( String file )
  {
    super( file );
  }

  /** gray method
  *  @param coeff the multipler to determine the
  *          color for each pixel
  *  @return a WritableImage
  */
  public WritableImage gray( double coeff )
  {
    int width = ( int ) getWidth( );
    int height = ( int ) getHeight( );
    WritableImage grayImage
         = new WritableImage( width, height );
    PixelWriter pw = grayImage.getPixelWriter( );
    PixelReader pr = getPixelReader( );

    for ( int x = 0; x < width; x++ )
    {
      for ( int y = 0; y < height; y++ )
      {
        Color currentColor = pr.getColor( x, y );
        double grayValue =  coeff *
          ( currentColor.getRed( ) + currentColor.getGre
            + currentColor.getBlue( ) );
        currentColor = Color.gray( grayValue );
        pw.setColor( x, y, currentColor );
      }
    }
    return grayImage;
  }
}
```

Description

A Constructor and Useful Methods of the *Image* Class

Package	**javafx.scene.image**
Constructor	
`Image( String URL )`	
constructs an *Image* from the file named in *URL*	
Return value	**Method name and argument list**
`PixelReader`	`getPixelReader( )`
returns a *PixelReader* object for accessing the pixels in the *Image*	
`double`	`getHeight( )`
returns the height of the image in pixels	
`double`	`getWidth( )`
returns the width of the image in pixels	

Description

The purpose of the *gray* method (lines 23–50) is to create an image composed of varying levels of gray. The gray level of each pixel in the new image is computed by multiplying the red, green, and blue components of the corresponding pixel in the original image by the parameter *coeff*.

Let's look at the *gray* method in detail. To get the size of the image, we call the *getWidth* and *getHeight* methods, shown in Table 17.15, which we have inherited from the *Image* class. Using this information, we create an empty *WritableImage* object of the same size. A *WritableImage* object can be used to create an image from pixels. **Table 17.16** shows the constructor and useful methods of the *WritableImage* class, which also inherits from the *Image* class.

A Constructor and a Useful Method of the *WritableImage* Class

Package	**javafx.scene.image**

Constructor	
`WritableImage( int width, int height )`	
constructs an empty WritableImage of the specified width and height	
Return value	**Method name and argument list**
`PixelWriter`	`getPixelWriter( )`
returns a *PixelWriter* object for writing the pixels in the *Image*	

Description

We then use the *getPixelWriter* factory method (line 34) to obtain a *PixelWriter* object to write pixel data into the *WritableImage.*

At line 35, we call the *getPixelReader* factory method to get a *PixelReader* object to read pixel data from the original image. A *PixelReader* object provides access to each pixel in the original image (see **Table 17.17**), while a *PixelWriter* object allows us to set the color of each pixel in the *WritableImage* (see **Table 17.18**).

A Method of the *PixelReader* Interface

Package	**javafx.scene.image**
Return value	**Method name and argument list**
`Color`	`getColor( int x, int y )`
returns the *Color* of the pixel at coordinate (*x*, *y*) in the image	

Description

A Method of the *PixelWriter Interface*

Package	**javafx.scene.image**

Return value	**Method name and argument list**
`void`	`setColor( int x, int y, Color c )`
sets the pixel at coordinate (*x*, *y*) to the color *c*	

Description

Using a nested *for* loop (lines 37–48), we use that *PixelReader* reference to move through each pixel in the original image. We first retrieve the color of each pixel by calling the *getColor* method of the *PixelReader*, passing it the current *x* and *y* values (lines 40–41).

Using the current color of the original image, we then multiply the sum of the red, green, and blue components by the *coeff* parameter (lines 42–44) and pass the resulting value to the *gray* method of the *Color* class (line 45); this returns a *Color* object where the red, green, and blue components have the *grayValue* parameter's value. The methods of the *Color* class that we use in this example are shown in **Table 17.19**. Finally, we use the *PixelWriter* to set the color of the corresponding pixel within the *WritableImage* (line 46) to the newly computed gray color.

Useful Methods of the *Color* Class

Package	**javafx.scene.paint**
Return value	**Method name and argument list**
`double`	`getRed( )`
returns the red component of the color in the range 0.0–1.0	
`double`	`getGreen( )`
returns the green component of the color in the range 0.0–1.0	
`double`	`getBlue( )`
returns the blue component of the color in the range 0.0–1.0	
`Color`	`gray( double value )`
returns a *Color* object where the red, green, and blue components are set to *value,* which can range from 0.0 to 1.0	

Description

We return the new image at line 49.

The Controller, *PhotoGrayerController.java,* shown in **Example 17.25**, handles any changes to the *Slider*'s value made by the user dragging the thumb in either direction. In the *initialize* method (lines 14–25), we instantiate the Model (line 16), sending the filename of the photo to the constructor.

Example 17.25 The Controller, *PhotoGrayerController.java*

```
 1 import javafx.beans.value.*;
 2 import javafx.event.*;
 3 import javafx.fxml.FXML;
 4 import javafx.scene.control.*;
 5 import javafx.scene.image.*;
 6
 7 public class PhotoGrayerController
 8 {
 9   private PhotoGrayer photoGrayer;
10
11   @FXML private ImageView grayImageView;
12   @FXML private Slider slider;
13
14   public void initialize( )
15   {
16     photoGrayer = new PhotoGrayer( "sushi.jpg" );
17
18     // initialize grayImageView
19     Image grayImage = photoGrayer.gray( slider.getValue( ) );
20     grayImageView.setImage( grayImage );
21
22     // set up event handling for slider
23     SliderHandler sh = new SliderHandler( );
24     slider.valueProperty( ).addListener( sh );
25   }
26
27   private class SliderHandler
28                implements ChangeListener<Number>
29   {
30     @Override
31     public void changed( ObservableValue<? extends Number> o,
32                          Number oldValue, Number newValue )
33     {
34       // update grayImageView
35       grayImageView.setImage(
36              photoGrayer.gray( newValue.doubleValue( ) ) );
37     }
38   }
39 }
```

Description

At lines 18 through 20, we initialize the second photo by calling the *gray* method in the Model, sending it the current slider value by calling the *getValue* method of the *Slider* class, as shown in **Table 17.20**.

Some Useful Methods of the *Slider* Class

Package	**javafx.scene.control**
Return value	**Method name and argument list**
`double`	`getValue( )`
returns the current value of the *Slider* as a *double*	
`DoubleProperty`	`valueProperty( )`
returns the current value of the *Slider* as a *DoubleProperty*, which is a *Property* wrapper class for a *double*, and implements the *ObservableValue<Number>* interface	

Description

The final initialization task is to register the event handler for the *Slider* control. Before we cover that, let's look at lines 27 through 38, where we define the event handler as a *private* **inner class** named *SliderHandler*. A *private* inner class is defined inside a *public* class and has access to all the members of the *public* class. Thus, declaring our event handler as a *private* inner class simplifies our code by giving the event handler direct access to our application's GUI components.

Similar to the *ComboBox*, for which we set up an *ObservableList*, the slider value is an *ObservableValue*, meaning that we can register an event handler that will be notified when the value changes. The *SliderHandler* class implements the *ChangeListener* interface, which has one method that we must override, shown in **Table 17.21** and here:

The Method of the *ChangeListener < T >* Interface

Package	**javafx.beans.value**
Return value	**Method name and argument list**
`void`	`changed( ObservableValue<? extends T> observable, T oldValue, T newValue)`

The *ChangeListener* must implement this method to handle changes to the *ObservableValue*.

Description

```
void changed( ObservableValue<? extends T> observable,
              T oldValue, T newValue )
```

Description

The syntax *<? extends T>* means that the observable value can be of any type that is a subclass of the generic type *T* or the type *T* itself. At lines 30 through 37, where we override the method above, we specify the *T* parameter to be *Number* (in *java.lang*), which is an *abstract* superclass extended by Java's numeric wrapper classes: *Byte*, *Short*, *Integer*, *Long*, *Float*, and *Double.* This works because our observable value is a *Double.*

Inside the method, we set the new color for the *ImageView grayImageView* by calling the *gray* method of the Model and sending it the new value of the *Slider*, obtained by calling the *doubleValue* method of the *Double* class, shown in **Table 17.22**.

A Useful Method of the *Double* Class

Package	**java.lang**
Return value	**Method name and argument list**
`double`	`doubleValue( )`

abstract method in the *Number* class overridden by the *Double* class; it returns the value of a *Double* object as a *double*

Description

Now we're ready to look at lines 23 and 24. At line 23, we instantiate an object of the *SliderHandler* inner class. At line 24, we call the *valueProperty* method of the *Slider* class (shown in Table 17.20), which returns the value of the *Slider* as a *DoubleProperty* object. Because the *DoubleProperty* class implements the *ObservableValue* interface, we can call the *addListener* method of that class to register our event handler. The *ObservableValue* interface, shown in **Table 17.23**, is also generic. The syntax *<? super T>* means that the observable value can be of any type that is a superclass of the generic type *T* or the type *T* itself.

The Methods of the *ObservableValue*<T> Interface

Package	**javafx.beans.value**
Return value	**Method name and argument list**
`void`	`addListener( ChangeListener<? super T> listener )`
adds a *ChangeListener* whose code will execute whenever the value of the *ObservableValue* changes	
`T`	`getValue( )`
returns the current value of the *ObservableValue* object	
`void`	`removeListener( ChangeListener<? super T> listener )`
removes the listener so that it is no longer notified of changes to the value of the *ObservableValue*	

Description

The last piece of the application is the launch class, shown in **Example 17.26**.

Example 17.26 *PhotoGraying.java*

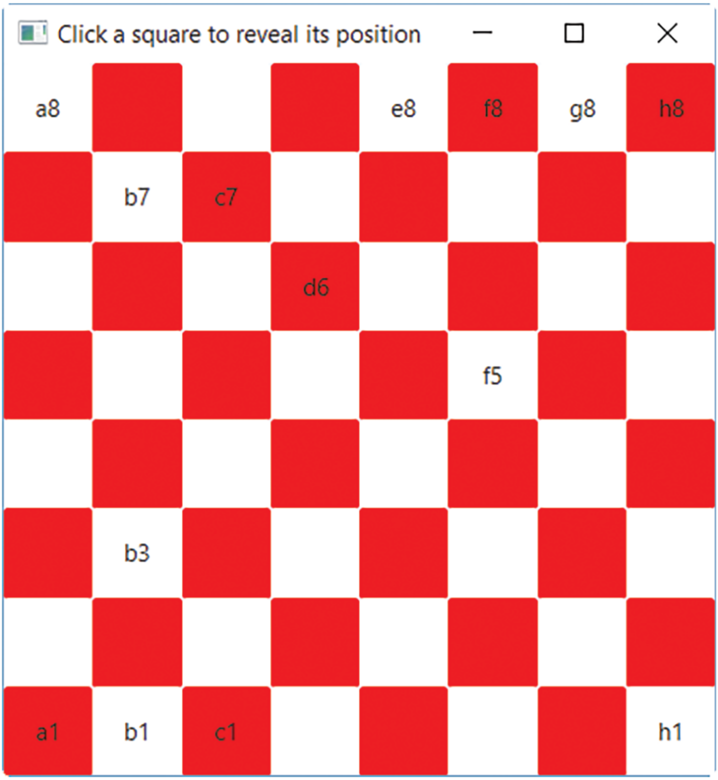
```
/*  Slider demo
    Anderson, Franceschi
*/

import java.net.URL;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.layout.VBox;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class PhotoGraying extends Application
{
  @Override
  public void start( Stage stage )
  {
    try
    {
      URL url =
        getClass( ).getResource( "fxml_photo_grayer.fxml" );
      VBox root = FXMLLoader.load( url );
      Scene scene = new Scene( root, 350, 320 );
      stage.setTitle( "Graying a photo" );
      stage.setScene( scene );
      stage.show( );
    }
    catch ( Exception e )
    {
      e.printStackTrace( );
    }
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

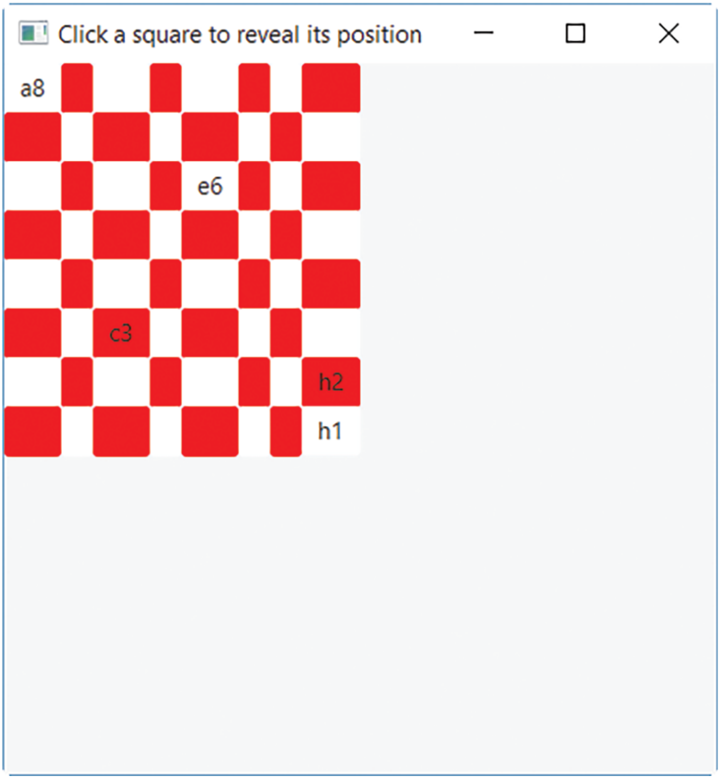
Description

with these end-of-chapter questions

17.20.1	Multiple Choice Exercises Questions 6, 7, 8, 9
17.20.2	Reading and Understanding Code Questions 21, 22, 23, 24, 25, 26, 27, 28, 29
17.20.3	Fill In the Code Questions 33, 34, 35

17.20.4	Identifying Errors in Code Question 52
17.20.5	Debugging Area—Using Messages from the Java Compiler and Java JVM Question 61
17.20.6	Write a Short Program Questions 64, 65, 66, 68, 69, 70, 71, 72, 75


Click a square to reveal its position
a8
f8
e8
g8
h8
b7
c7
d6
f5
b3
a1
b1
c1
h1



17.10 Building a GUI Programmatically

We have been defining GUIs using FXML, which works well when we know the number and type of controls in our GUI. Sometimes, however, the GUI is dynamic and because we may not know how many buttons or other controls we need until runtime, we must define the GUI, that is, instantiate the controls and set their properties, programmatically. For example, the GUI may include buttons that represent URL links stored in a file or on a web server. In that case, the number of buttons can vary each time we run our application. In other applications, it may be more convenient to build the GUI by code rather than with FXML. For example, if the GUI represents a tic-tac-toe game or a chessboard, it is convenient to represent the GUI as a two-dimensional array of buttons, which we define and instantiate programmatically.

We illustrate this approach with an example where we build a chessboard and reveal the position of a square as its column letter and row number when the user clicks on it. **Figure 17.12** shows the application running after the user has clicked on several buttons.

Figure 17.12 Running the Chessboard Example

Description

Before defining the View and the Controller, we define our Model, the *BoardGame* class, shown in **Example 17.27**. We define a board with an array of *chars* representing the column letters and an array of *ints* representing the row numbers. We declare two instance variables for these arrays at lines 12 and 13. We include an array of colors (line 14) so that clients of this class can use custom colors for the board. The *getSquareText* method (lines 29–37) returns a *String* representation of a square on the board as the concatenation of the letter and digit at the row and column indexes requested by the two parameters.

Example 17.27 The *BoardGame* Class

```java
/*  BoardGame class
    Anderson, Franceschi
*/

import javafx.scene.paint.Color;

public class BoardGame
{
  private static char [ ] hexDigits =
         { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
           'A', 'B', 'C', 'D', 'E', 'F' };
  private int [ ] rows;
  private char [ ] columns;
  private Color [ ] colors; // to color the rows and columns

  /** Overloaded Constructor
  *  @param newRows     the row numbers
  *  @param newColumns  the column letters
  *  @param newColors   the colors
  */
  public BoardGame( int [ ] newRows, char [ ] newColumns,
                    Color [ ] newColors )
  {
    rows = newRows;
    columns = newColumns;
    colors = newColors;
  }

  /** getSquareText method
  *  @param row      the row index
  *  @param column   the column index
  *  @return a String, a concatenation of columns[column] and rows[row]
  */
  public String getSquareText( int row, int column )
  {
    return String.valueOf( columns[column] ) + rows[row];
  }

  /** getSquareColor method
  *  @param row      the row index
  *  @param column   the column index
  *  @return a String, a hex representation of
  *                    colors[(row + col) % colors.length]
  */
  public String getSquareColor( int row, int col )
  {
    Color squareColor = colors[(row + col) % colors.length];
    return toHexString( squareColor );
  }

  /** toHex utility method
  *  @param colorIntensity   a double between 0 and 1 inclusive
  *  @return a String, the hex representation of colorIntensity times 255
  */
  public String toHex( double colorIntensity )
  {
    int colorIntensityAsInt = ( int ) ( colorIntensity * 255 );
    if ( colorIntensityAsInt > 255 )
      colorIntensityAsInt = 255;
    else if ( colorIntensityAsInt < 0 )
      colorIntensityAsInt = 0;

    int firstDigit = colorIntensityAsInt / 16;
    int secondDigit = colorIntensityAsInt % 16;

    return String.valueOf( hexDigits[firstDigit] ) + hexDigits[secondDigit];
  }

  /** toHexString method
  *  @param color  a Color
  *  @return a String, a hex representation of color
  */
  public String toHexString( Color color )
  {
    String colorText = "#" + toHex( color.getRed( ) )
                           + toHex( color.getGreen( ) )
                           + toHex( color.getBlue( ) );
    return colorText;
  }

  /** getNumberOfRows method
  *  @return the number of rows
  */
  public int getNumberOfRows( )
  {
    return rows.length;
  }

  /** getNumberOfColumns method
  *  @return the number of columns
  */
  public int getNumberOfColumns( )
  {
    return columns.length;
  }
}
```

Description

The *getSquareColor* method (lines 39–49) returns a hexadecimal representation of the color of the square on the board given its row and column numbers, which are sent as parameters to the method. This method can be used to color the board with alternating colors retrieved from the array *colors*.

We provide accessors for the number of rows and the number of columns (lines 81–87 and 89–95).

Even though we are defining the GUI by code, it is possible to separate the View class from the Controller class, as in the previous examples. However, in order to keep this example simple, we put the View and the Controller in the same class.

Regardless of the layout container used, the constructor of our application needs to perform the following operations:

- Instantiate components
- Add components to the layout container

For this example, we use a *GridPane* layout container that allows us to arrange components in a grid. We can visualize a *GridPane* as a table made up of cells in rows and columns. Each cell can contain one component. These cells can have different sizes. In this application, as in many, however, each cell has the same size. Components are placed on the grid at a specified column and row using one of the *add* methods of the *GridPane* class, such as the one shown in **Table 17.24**. Note that the column parameter is given before the row parameter.

Useful Methods of the *GridPane* Class

Package	**javafx.scene.layout**
Return value	**Method name and argument list**
`void`	`add( Node child, int columnIndex, int rowIndex )`
adds *child* to this *GridPane* at row *rowIndex* and column *columnIndex*	
`ObservableList<RowConstraints>`	`getRowConstraints( )`

returns a list of row constraints for this *GridPane*	
`ObservableList<ColumnConstraints>`	`getColumnConstraints( )`
returns a list of column constraints for this *GridPane*	

Description

By default, the rows and columns in the grid are sized to their preferred size, which is based on the content of each cell and independent of the size of the *GridPane*. A *GridPane* maintains lists of row and column constraints that it uses to size each cell in the grid. There are no row or column constraints when a *GridPane* is first instantiated, so these two lists are originally empty. If we want to control the height of each row in the grid, we add as many row constraints as there are rows in the grid; similarly, if we want to control the width of each column in the grid, we add as many column constraints as there are columns in the grid. In this example, we want all the rows to have the same height and all the columns to have the same width. We also want all the cells to fill the space in the *GridPane*.

The *getRowConstraints* and *getColumnConstraints* methods of the *GridPane* class, shown in Table 17.24, retrieve the list of row and column constraints, respectively.

We use the *add* method of the *ObservableList* interface, inherited from the *List* interface, to add a row or a column constraint to the appropriate list. When we add a constraint to the list of row or column constraints, it is automatically added to the list of constraints in the *GridPane*. This method is shown in **Table 17.25**.

The *add* and *clear* Methods of the *List* Interface

Package	**java.util**
Return value	**Method name and argument list**
`boolean`	`add( E e )`
appends *e* to the end of the list	
`void`	`clear( )`

removes all the elements from the list

Description

The *RowConstraints* and *ColumnConstraints* classes can be used to define the height of a row and width of a column in a *GridPane*, respectively. Often, we want to set the dimensions of a row or column as a percentage of the available space within the *GridPane*. We do this with the *setPercentHeight* and *setPercentWidth* methods of the *RowConstraints* and *ColumnConstraints* classes. These methods are shown in **Tables 17.26** and **17.27**.

The *setPercentHeight* Method of the *RowConstraints*

Package	**javafx.scene.layout**
Return value	**Method name and argument list**
`void`	`setPercentHeight( double value )`

sets the height of a row as a percentage of the total height of the *GridPane*

Description

The *setPercentWidth* Method of the *ColumnConstraints* Class

Package	**javafx.scene.layout**
Return value	**Method name and argument list**
`void`	`setPercentWidth( double value )`

sets the width of a column as a percentage of the total width of the

Description

Example 17.28 shows how to define a GUI by code using a *GridPane* to display a chessboard. Each position on a chessboard is identified by a letter (a–h) and a number (1–8). From the standpoint of the white player, the lower left square is a1, and from the standpoint of the black player, the lower left square is h8. When the user clicks on a square on the chessboard, our application displays its position.

Example 17.28 The *BoardView* Class

```
/*  Using GridPane to organize our window
    Anderson, Franceschi
*/

import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.control.Button;
import javafx.scene.layout.*;

public class BoardView extends GridPane
{
  private BoardGame game;
  private Button [ ][ ] squares;

  public BoardView( BoardGame newGame )
  {
    super( );
    game = newGame;

    // set up grid according to Model
    int rows = game.getNumberOfRows( );
    int columns = game.getNumberOfColumns( );
    RowConstraints row = new RowConstraints( );
    row.setPercentHeight( 100.0 / rows );
    ColumnConstraints col = new ColumnConstraints( );
    col.setPercentWidth( 100.0 / columns );
    for ( int i = 0; i < rows; i++ )
      getRowConstraints( ).add( row );
    for ( int j = 0; j < columns; j++ )
      getColumnConstraints( ).add( col );

    squares = new Button[rows][columns];
    ButtonHandler bh = new ButtonHandler( );

    for ( int i = 0; i < rows; i++ )
    {
      for ( int j = 0; j < columns; j++ )
      {
        // instantiate the button with no text
        squares[i][j] = new Button( );

        // color the button
        squares[i][j].setStyle( "-fx-background-color:"
          + game.getSquareColor( i, j ) );

        // add the button
        add( squares[i][j], j, i );

        // make button fill up available width and height
        squares[i][j].setMaxWidth( Double.MAX_VALUE );
        squares[i][j].setMaxHeight( Double.MAX_VALUE );

        // register listener on button
        squares[i][j].setOnAction( bh );
      }
    }
  }

  // private inner class event handler
  private class ButtonHandler implements EventHandler<ActionEvent>
  {
    @Override
    public void handle( ActionEvent event )
    {
      for ( int i = 0; i < squares.length; i++ )
      {
        for ( int j = 0; j < squares[i].length; j++ )
        {
          if ( event.getSource( ) == squares[i][j] )
          {
            squares[i][j].setText( game.getSquareText( i, j ) );
            return;
          }
        }
      }
    }
  }
}
```

Description

The *BoardView* class, shown in Example 17.28, extends *GridPane*. In this way, we can set a *BoardView* object as the root of the scene that we place on the stage.

A two-dimensional array of *Buttons*, named *squares* (line 13), makes up the chessboard. We declare an instance variable of type *BoardGame,* our Model, at line 12. The constructor (lines 15–57) defines the GUI and sets up event handling. We call the constructor of *GridPane* at line 17 and retrieve the number of rows and columns from the Model at lines 20 through 22. At lines 23 through 30, we define and add row and column constraints. After instantiating a *RowConstraints* object at line 23, we set its percentage height to 100%, divided by the number of rows. We then add that row constraint to the list of row constraints as many times as there are rows (lines 27–28). In this way, each row has the same height. We define the column constraints similarly, so that each column has the same width.

At line 33, we declare and instantiate the listener *bh*, a *ButtonHandler* object reference. Because we are interested in events related to buttons, our *ButtonHandler private* inner class implements the *EventHandler* interface and overrides the *handle* method. At line 32, we instantiate the two-dimensional array *squares*. We have a two-dimensional array of *Buttons*, so we use nested *for* loops at lines 35 through 56 to instantiate the *Buttons*, add them to the *GridPane*, and register the listener on all the buttons. Using FXML, to include a *Button* in a layout container, we define a *Button* element and specify attributes for that *Button* element in an *.fxml* file. To create and define a *Button* programmatically, we instantiate the *Button* using a constructor and call various methods to set its properties. These setter methods typically accept a parameter that represents the new value for the *Button* property. Some of these methods are shown in **Table 17.28**. Remember from Figure 17.2 that the *Button* class is a subclass of *ButtonBase*, *Labeled*, *Control*, *Region*, *Parent*, *Node*, and *Object*, and inherits some properties from its superclasses.

Selected Methods of and Inherited by the *Button*

Class

Package	**javafx.scene.control**
Constructor	
`Button( )`	
creates an empty Button	
Return value	**Method name and argument list**
`void`	`setStyle( String style )`
sets the value of the *style* property of this *Button* to *style*, which is expected to be a CSS-like *String*. This method is inherited from *Node*.	
`void`	`setMaxWidth( double value )`
sets the value of the *maxWidth* property of this *Button* to *value*. This method is inherited from *Region*.	
`void`	`setMaxHeight( double value )`
sets the value of the *maxHeight* property of this *Button* to *value*. This method is inherited from *Region*.	
`void`	`setOnAction( EventHandler<ActionEvent> handler )`
registers *handler* as the event handler for this *Button*. When the button is clicked, the *handle* method of handler's class will be called automatically. This method is inherited from *ButtonBase*.	

Description

At lines 42 through 44, we color the squares according to the coloring pattern and colors set in the Model. In order to do this, we call the *setStyle* method of the *Button* class, inherited from *Node*, and also shown in Table 17.28. The *setStyle* method accepts a CSS-like *String* parameter composed of a semicolon-separated list of attribute-value pairs, with each attribute and value separated by a colon (:). The format of the CSS-style *String* parameter is as follows:

```
"styleAttribute1:value1;styleAttribute2:value2;styleAttribute3:value3; ..."
```

Description

For example, if we want to set the background color to blue and the text's font size to 25, we could use the following *String* as the parameter to the *setStyle* method:

```
"-fx-background-color:blue;-fx-font-size:25"
```

Description

At lines 49 through 51, we ensure that each *Button* will fill the available space within the *GridPane* by setting its maximum width and height to the maximum possible value by calling the *setMaxWidth* and *setMaxHeight* methods (see Table 17.28), using the *static* constant of the *Double* wrapper class (*Double.MAX_VALUE*) that represents the maximum value that a *double* data type can hold. If we remove these statements, the *Buttons* will be sized to their default size, which is based on their contents: in this case, the button's text and the font size of that text. At lines 53 and 54, we call the *setOnAction* method (see Table 17.28) to register the *ButtonHandler bh* on each *Button*.

REFERENCE POINT

For a complete listing of JavaFX style attributes, see *https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html*.

In the *handle* method, we use nested *for* loops at lines 65 through 75 to identify the source of the event, that is, which button the user clicked. We then set the text of that button to its board position (line 71), retrieving the corresponding value from the Model. Having found the source of the event, we then exit the event handler via the *return* statement (line 72) to interrupt the *for* loops and thus avoid unnecessary processing.

Finally, the *ChessBoard* class, shown in **Example 17.29**, includes the *main* and *start* methods to create a *BoardView* showing a *BoardGame* game. The *letters*, *digits*, and *boardColors* arrays (lines 14–16) define the board. Try changing either the *letters*, *digits*, or *boardColors* array and running the application again. The grid will be resized correctly and with new colors, showing the reusability of our classes.

Example 17.29 The *ChessBoard* Class

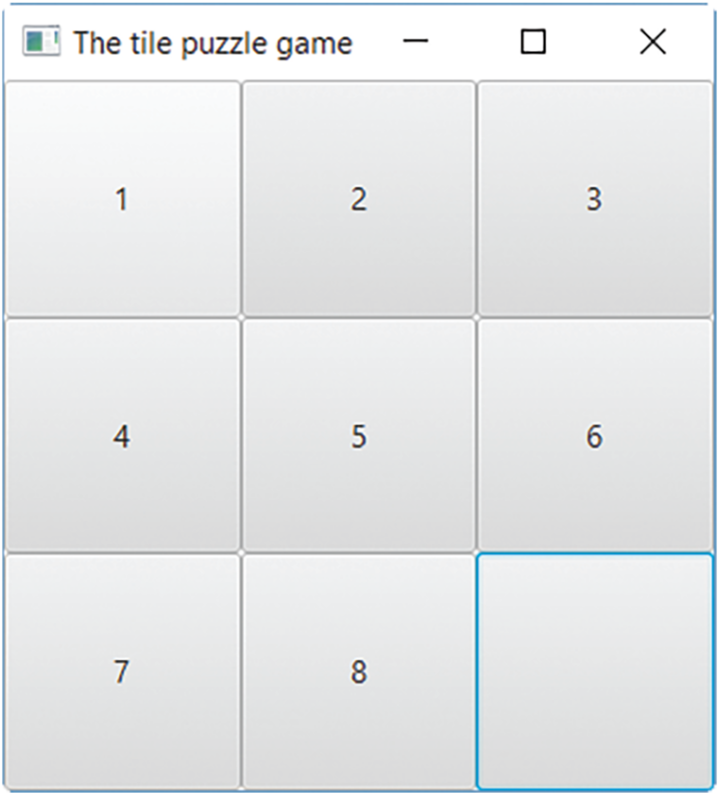
```
/*  ChessBoard class
    Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.scene.paint.Color;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class ChessBoard extends Application
{
  public void start( Stage stage )
  {
    char [ ] letters = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' };
    int [ ] digits = { 8, 7, 6, 5, 4, 3, 2, 1 };
    Color [ ] boardColors = { Color.WHITE, Color.RED };

    BoardGame game = new BoardGame( digits, letters, boardColors );
    BoardView view = new BoardView( game );

    Scene scene = new Scene( view, 450, 450 );
    stage.setScene( scene );
    stage.setTitle( "Click a square to reveal its position" );
    stage.show( );
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

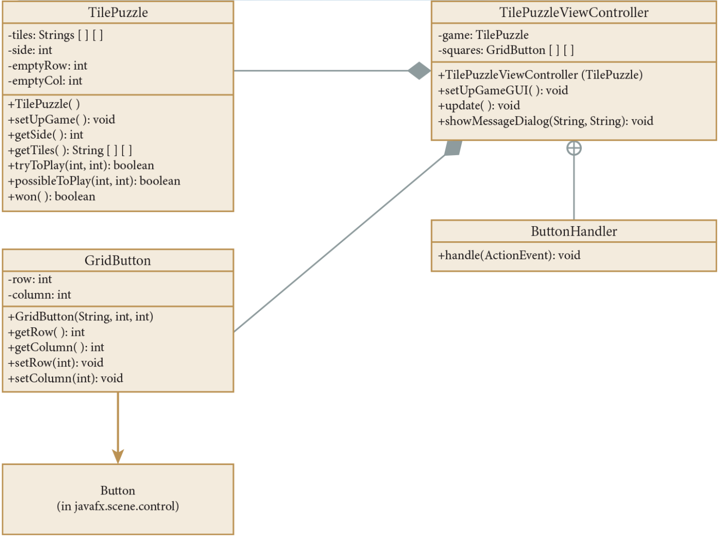
Description

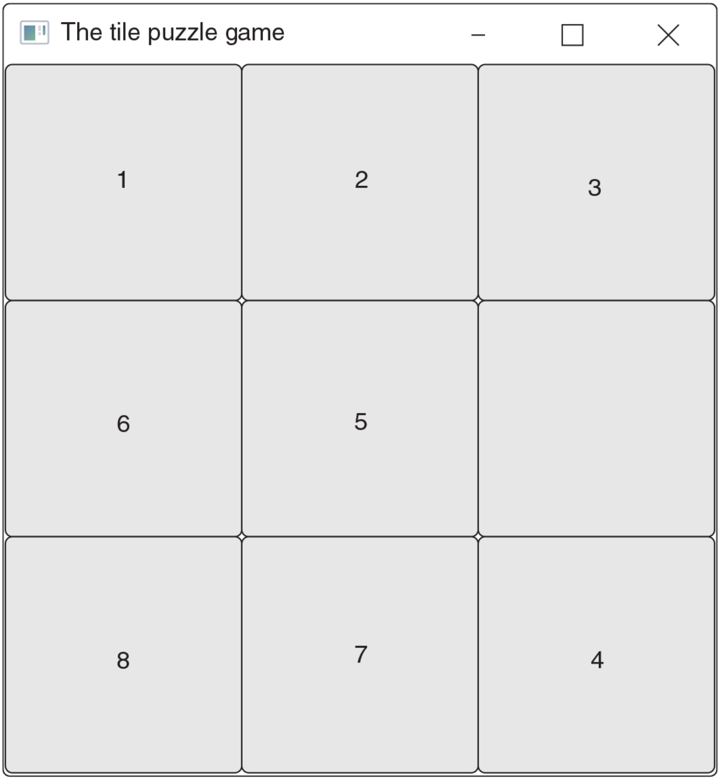
Figure 17.13 shows the application if we eliminate the row and column constraints by commenting out lines 23 through 30 of Example 17.28. Similarly, we will have the same result if we comment out lines 49 through 51, because the buttons will have their computed preferred size, which is based on their content. As we see, the cells do not fill up the width and height of the *GridPane*; the buttons have their default minimum size because they are empty. When we click on a button, it expands in width to display its new text, and the width of the whole column increases accordingly.

Figure 17.13 Running the Chessboard Example Without Row and Column Constraints

Description


The tile puzzle game
1
2
3
4
5
6
7
8


TilePuzzle
-tiles: Strings [ ] [ ]
-side: int
-emptyRow: int
-emptyCol: int
+TilePuzzle( )
+setUpGame( ): void
+getSide( ): int
+getTiles( ): String [ ] [ ]
+tryToPlay(int, int): boolean
+possibleToPlay(int, int): boolean
+won( ): boolean
TilePuzzleViewController
-game: TilePuzzle
-squares: GridButton [ ] [ ]
+TilePuzzleViewController (TilePuzzle)
+setUpGameGUI( ): void
+update( ): void
+showMessageDialog(String, String): void
ButtonHandler
+handle(ActionEvent): void
GridButton
-row: int
-column: int
+GridButton(String, int, int)
+getRow( ): int
+getColumn( ): int
+setRow(int): void
+setColumn(int): void
Button
(in javafx.scene.control)



17.11 Layout Containers: Dynamically Setting Up the GUI Using *GridPane*

Layout managers can be set dynamically, based on user input. For example, the user could enter the number of rows or columns of the grid. Based on user input, we can also rearrange the components using another layout container, such as *HBox*. The user could also instruct us to remove components and add others. Our next example, the Tile Puzzle game, will illustrate some of these capabilities.

In the Tile Puzzle game, eight tiles displaying the digits 1 through 8 are scrambled on a 3-by-3 grid, leaving one cell empty. Any tile adjacent to the empty cell can be moved to the empty cell by clicking on the numbered tile. The goal is to rearrange the tiles so that the numbers are in the correct order, as shown in **Figure 17.14**.

Figure 17.14 The Winning Position of a 3-by-3 Tile Puzzle Game

Description

The Tile Puzzle game can also be played on a 4-by-4, 5-by-5, and more generally, *n*-by-*n* grid. In this example, we set up a 3-by-3 grid for the first game and then randomly select a 3-by-3, 4-by-4, 5-by-5, or 6-by-6 grid for subsequent games. Later, we can modify this example to allow the user to specify the size of the grid.

Before designing the GUI class, we first code the Model, a class to encapsulate the functionality of the tile puzzle game; the *TilePuzzle* class (**Example 17.30**) handles the creation of a game of a given size, enables play, and enforces the rules of the game.

Example 17.30 The *TilePuzzle* Class

```java
/** TilePuzzle class
*   Anderson, Franceschi
*/

public class TilePuzzle
{
 private String [ ][ ] tiles;
 private int side; // grid size
 private int emptyRow;
 private int emptyCol;

 /** constructor
 * @param newSide grid size
 */
 public TilePuzzle( int newSide )
 {
  setUpGame( newSide );
 }

 /** setUpGame
 * @param newSide grid size
 */
 public void setUpGame( int newSide )
 {
  if ( newSide < 3 )
   side = 3;
  else
   side = newSide;
  emptyRow = side - 1;
  emptyCol = side - 1;
  tiles = new String[side][side];

  // initialize tiles
  for ( int i = 0; i < side; i++ )
  {
   for ( int j = 0; j < side; j++ )
   {
    tiles[i][j] = String.valueOf( ( side * side )
                                - ( side * i + j + 1 ) );
   }
  }
  // set empty cell label to blank
  tiles[side - 1][side - 1] = "";
 }

 /** getSide
 * @return side
 */
 public int getSide( )
 {
   return side;
 }

 /** getTiles
 * @return a copy of tiles
 */
 public String[ ][ ] getTiles( )
 {
  String[ ][ ] copyOfTiles = new String[side][side];

  for ( int i = 0; i < side; i++ )
  {
   for ( int j = 0; j < side; j++ )
   {
    copyOfTiles[i][j] = tiles[i][j];
   }
  }
  return copyOfTiles;
 }

 /** tryToPlay
 * enable play if play is legal
 * @return true if the play is legal, false otherwise
 */
 public boolean tryToPlay( int row, int col )
 {
  if ( possibleToPlay( row, col ) )
  {
   // play: switch empty String and tile label at row, col
   tiles[emptyRow][emptyCol] = tiles[row][col];
   tiles[row][col] = "";
   // update emptyRow and emptyCol
   emptyRow = row;
   emptyCol = col;
   return true;
  }
  else
   return false;
 }

 /** possibleToPlay
 * @return true if the play is legal, false otherwise
 */
 public boolean possibleToPlay( int row, int col )
 {
  if ( ( col == emptyCol && Math.abs( row - emptyRow ) == 1 )
         || ( row == emptyRow && Math.abs( col - emptyCol ) == 1 ) )
   return true;
  else
   return false;
 }

 /** won
 * @return true if correct tile order, false otherwise
 */
 public boolean won( )
 {
  for ( int i = 0; i < side ; i++ )
  {
   for ( int j = 0; j < side; j++ )
   {
    if ( !( tiles[i][j].equals(
            String.valueOf( i * side + j + 1 ) ) )
       && ( i != side - 1 || j != side - 1 ) )
       return false;
   }
  }
  return true;
 }
}
```

Description

The instance variable *tiles* (line 7), a two-dimensional array of *Strings*, stores the state of the puzzle. Each element of *tiles* represents a cell in the puzzle grid. The instance variable *side*, declared at line 8, holds the size of the grid. The instance variables *emptyRow* and *emptyCol*, declared at lines 9 and 10, identify the empty cell in the puzzle grid.

The constructor (lines 12–18) calls the *setUpGame* method (lines 20–44), passing the size of the grid (*newSide*) as an argument. The controller also calls the *setUpGame* method before starting each new game.

Inside the *setUpGame* method, we assign *newSide* to *side*. Rather than randomly generating each tile label, which would complicate this example, we assign the labels to the tiles in descending order using nested *for* loops (lines 33–41). We set the empty cell to the last cell in the grid (lines 42 and 43).

The *tryToPlay* method (lines 71–89) first checks if the play is legal by calling the *possibleToPlay* method (line 77). The play is legal if the tile the user clicked is next to the empty tile. If the *possibleToPlay* method returns *true*, we proceed with the play and return *true*; otherwise, we return *false*. Playing means swapping the values of the empty cell (*emptyRow*, *emptyCol*) and the cell that was just played, represented by the two parameters of the *tryToPlay* method, *row* and *col*.

The *possibleToPlay* method is coded at lines 91 through 101. If the play is legal—that is, if the tile played is within one cell of the empty cell—the method returns *true*; otherwise, the method returns *false*.

The *won* method (lines 103–119) checks if the tiles are in order. If so, the *won* method returns *true*; otherwise, the method returns *false*.

As in the chessboard example, when the user clicks on a button, we need to identify the row and the column of that button. In order to avoid looping through all the buttons, we create a class that *extends* the *Button* class and we add two instance variables for the row and column indexes. In this way, when a button belonging to a two-dimensional array of buttons is clicked, we can access its *row* and *column* instance variables in order to identify which button was clicked.

Example 17.31 shows the *GridButton* class, which extends the *Button* class and adds two instance variables, *row* and *column* (lines 9–10). The constructor at lines 12 through 22 allows the client to create a *GridButton* object with a specified text, row, and column. Note that this class is not specific to the tile puzzle application and can be reused for other applications involving an array of buttons placed on a grid.

Example 17.31 The *GridButton* Class

```java
/** GridButton class
*   Anderson, Franceschi
*/

import javafx.scene.control.Button;

public class GridButton extends Button
{
 private int row;
 private int column;

 /** Constructor
 * @param title text for button
 * @param newRow row
 * @param newColumn column
 */
 public GridButton( String title, int newRow, int newColumn
 {
  super( title );
  setRow( newRow );
  setColumn( newColumn );
 }

 /** getRow method, accessor for row
 * @return row
 */
 public int getRow( )
 {
   return row;
 }

 /** getColumn method, accessor for column
 * @return column
 */
 public int getColumn( )
 {
   return column;
 }

 /** setRow method, mutator for row
 * @param row, new value for row
 */
 public void setRow( int row )
 {
   if ( row >= 0 )
     this.row = row;
 }

 /** setColumn method
 * @param column, new value for column
 */
 public void setColumn( int column )
 {
   if ( column >= 0 )
     this.column = column;
 }
}
```

Description

Figure 17.15 shows the UML diagram for the *TilePuzzle* and *TilePuzzleViewController* classes.

Figure 17.15 The UML Diagram for *TilePuzzle* and *TilePuzzleViewController*

Description

Example 17.32 shows the *TilePuzzleViewController* class; as in the *BoardView* class in the previous example, it is a subclass of *GridPane* (line 12) and combines the View and the Controller.

Example 17.32 The *TilePuzzleViewController* Class

```java
/* Using GridPane dynamically
   Anderson, Franceschi
*/

import java.util.Random;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.*;
import javafx.scene.layout.*;

public class TilePuzzleViewController extends GridPane
{
  private GridButton [ ][ ] squares;
  private TilePuzzle game; // the tile puzzle game

  public TilePuzzleViewController( TilePuzzle newGame )
  {
    super( );
    game = newGame;
    setUpGameGUI( );
  }

  public void setUpGameGUI( )
  {
    // remove all components and constraints
    getChildren( ).clear( );
    getRowConstraints( ).clear( );
    getColumnConstraints( ).clear( );

    // set up grid constraints
    RowConstraints row = new RowConstraints( );
    row.setPercentHeight( 100.0 / game.getSide( ) );
    ColumnConstraints col = new ColumnConstraints( );
    col.setPercentWidth( 100.0 / game.getSide( ) );
    for ( int i = 0; i < game.getSide( ); i++ )
      getRowConstraints( ).add( row );
    for ( int j = 0; j < game.getSide( ); j++ )
      getColumnConstraints( ).addAll( col );

    squares = new GridButton [game.getSide( )][game.getSide( )];
    ButtonHandler bh = new ButtonHandler( );

    for ( int i = 0; i < game.getSide( ); i++ )
    {
      for ( int j = 0; j < game.getSide( ); j++ )
      {
        squares[i][j] = new GridButton( game.getTiles( )[i][j], i, j );

        // add the button
        add( squares[i][j], j, i );

        // make button fill up available width and height
        squares[i][j].setMaxWidth( Double.MAX_VALUE );
        squares[i][j].setMaxHeight( Double.MAX_VALUE );

        // register listener on button
        squares[i][j].setOnAction( bh );
      }
    }
  }

  public void update( )
  {
    for ( int i = 0; i < game.getSide( ); i++ )
      for ( int j = 0; j < game.getSide( ); j++ )
         squares[i][j].setText( game.getTiles( )[i][j] );

    if ( game.won( ) )
    {
      showMessageDialog( "Congratulations",
                         "You won\nSetting up a new game" );
      Random random = new Random( );
      int sideOfPuzzle = 3 + random.nextInt( 4 );
      game.setUpGame( sideOfPuzzle );
      setUpGameGUI( );
    }
  }

  public void showMessageDialog( String title, String message )
  {
    Alert alert = new Alert( AlertType.INFORMATION );
    alert.setTitle( title );
    alert.setHeaderText( "" );
    alert.setContentText( message );
    alert.showAndWait( );
  }

  // private inner class event handler
  private class ButtonHandler implements EventHandler<ActionEvent>
  {
    @Override
    public void handle( ActionEvent event )
    {
      GridButton button = ( GridButton ) event.getSource( );
      if ( game.tryToPlay( button.getRow( ), button.getColumn( ) ) )
        update( );
    }
  }
}
```

Description

Each tile in the game is a *GridButton*. The instance variable *squares* (line 14) holds a two-dimensional array of *GridButtons* so that each element of *squares* is a cell in the game grid. The *TilePuzzle* instance variable *game*, declared at line 15, represents the Model. When the user plays, we call the various methods of the *TilePuzzle* class to enforce the rules of the game.

The constructor (lines 17–22) calls the constructor of *GridPane*, instantiates *game,* and calls the *setUpGameGUI* method. The constructor and the *setUpGameGUI* method define the View. The *private* inner class *ButtonHandler* (lines 90–100) and the *update* method (lines 63–78) make up the Controller.

The *setUpGameGUI* method (lines 24–61) displays the game in its starting position. We first remove all the components and the row and column constraints from this *GridPane* (lines 26–29). In order to access the children of this *GridPane*, we call the *getChildren* method inherited from *Pane* and shown in **Table 17.29**. It returns an *ObservableList* of *Nodes* that are the children of this *GridPane*. With it, we call the *clear* method that *ObservableList* inherits from the *List* interface. That removes all the *Buttons* from this *GridPane*. Because the next game may be 3-by-3, 4-by-4, or 5-by-5 grids, we also need to remove the current row and column constraints of this *GridPane*. We access them using the *getRowConstraints* and *getColumnConstraints* methods of the *GridPane* class, shown earlier in Table 17.24. They also return a list of *Observable* objects, *RowConstraints,* and *ColumnConstraints*, respectively. We remove these constraints by calling the *clear* method, shown in Table 17.25. We then reset the row and column constraints of this *GridPane* layout container at lines 31 through 39. We instantiate the *squares* array and our event handler at lines 41 and 42. After that, we use nested *for* loops (lines 44–60) to instantiate each button, add it to the container, force the button to fill the available width and height, and register the event handler. The *squares* array parallels the *tiles* array of the *TilePuzzle* class; the *squares* buttons are labeled with the values of *tiles*.

The *getChildren* Method of the *GridPane* Class Inherited from the *Pane* Class

Package	**javafx.scene.layout**
Return value	**Method name and argument list**
`ObservableList<Node>`	`getChildren( )`
returns the children of this pane as an *ObservableList* of *Nodes*	

Description

When a button is clicked, our *ButtonHandler handle* method is executed. To determine which button was clicked, we call the *getSource* method using the *ActionEvent* event object that was sent to the method (line 96). Because the *getSource* method returns an *Object*, we need to type cast the return value to a *GridButton*. We then need to determine whether the button clicked was a legal play. To do this, we call the *tryToPlay* method at line 97 with the *row* and *column* instance variable values of the button. If that method returns *true*, the play was legal and the model has changed (i.e., a tile has been moved). We then call the *update* method to make the view (i.e., the buttons) reflect the model (line 98).

The *update* method first updates the *squares* button array (lines 65–67). We then test if the current move solved the puzzle by calling the *won* method (line 69) of the *TilePuzzle* class. If the *won* method returns *true*, we congratulate the user by popping up a dialog box at lines 71 and 72.

The *Alert* class, part of the *javafx.scene.control* package, enables us to display one of many predefined dialog boxes. A constructor and several methods are shown in **Table 17.30**. Most methods of *Alert* are inherited from its superclass, *Dialog*. The constructor accepts an *Alert.AlertType* parameter that defines the type of dialog box that we want to display. *Alert.AlertType*, a *static* inner class of *Alert*, is an *enum* that defines several constants that can be used to configure the icon and style of the dialog box. The *INFORMATION* type is used to configure the dialog box so that it just informs the user of something. Other types are CONFIRMATION, ERROR, WARNING, and NONE. At line 82 of the *showMessageDialog* method (lines 80–87), we instantiate a dialog box using the *INFORMATION* type. We then set the contents of the dialog box at lines 83–85. Note that if we did not set the header text of the dialog box to an empty *String* at line 84, the dialog box would be bigger and look a bit awkward. Finally, we display the dialog box at line 86 by calling the *showAndWait* method.

Constructor and Useful Methods of the *Alert* Class

Package	**javafx.scene.control**
Constructor	
`Alert( Alert.AlertType type )`	
constructs an *Alert* of type *type*.	
Return value	**Method name and argument list**
`void`	`setTitle( String title )`
sets the title of this dialog box.	
`void`	`setHeaderText( String message )`
sets the text to display in the header of this dialog box. If *message* is the empty *String*, no space is allocated for the header and the dialog box will be smaller.	
`void`	`setContentText( String message )`
sets the text to display inside dialog box.	
`Optional<R>`	`showAndWait( )`
shows this dialog box and waits for the user's response.	

Description

We then randomly generate a grid size between three and six (lines 73–74) for the next game and call the *setUpGame* method (line 75) and the *setUpGameGUI* method (line 76) to begin a new game with the new grid size.

Finally, the *PlayTilePuzzle* class, shown in **Example 17.33**, includes the *main* and *start* methods to create a *TilePuzzleViewController* application showing a *TilePuzzle* game. **Figure 17.16** shows a run of this game after the user has moved some tiles.

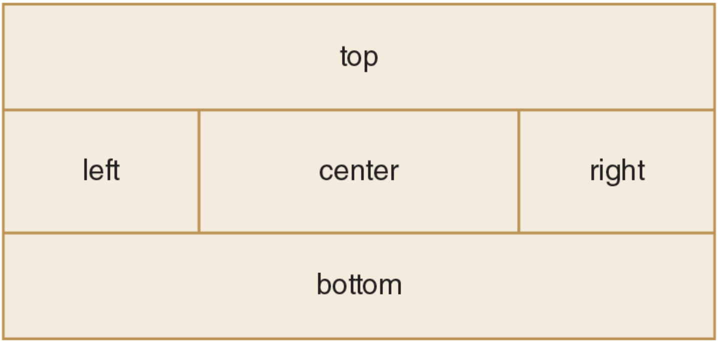
Example 17.33 The *PlayTilePuzzle* Class

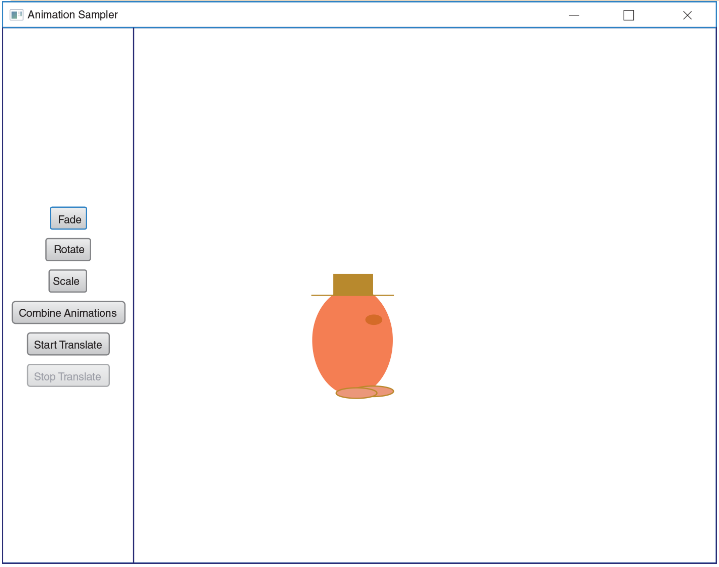
```
/*  PlayTilePuzzle class
    Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class PlayTilePuzzle extends Application
{

  @Override
  public void start( Stage stage )
  {
    TilePuzzle puzzle= new TilePuzzle( 3 );
    TilePuzzleViewController root
      = new TilePuzzleViewController( puzzle );

    Scene scene = new Scene( root, 350, 350 );
    stage.setTitle( "The tile puzzle game" );
    stage.setScene( scene );
    stage.show( );
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description

Figure 17.16 The Tile Puzzle Game in Progress

Description


top
left
center
right
bottom


Animation Sampler
Fade
Rotate
Scale
Combine Animations
Start Translate
Stop Translate

17.12 *BorderPane* Layout, Animations, Sounds, and Lambda Expressions

We have a lot to talk about in this section. We introduce a new layout class, *BorderPane*. We demonstrate animating nodes with some accompanying sound. And we explain using lambda expressions for event handlers.

A *BorderPane* layout container organizes its nodes into five positions: *top*, *bottom*, *left*, *right*, and *center*, with each position holding, at most, one node. The size of each position expands or contracts depending on the size of the node in that position, the sizes of the nodes in the other positions, and whether the other positions contain a node. The *center* position expands to fill any remaining space; if not enough space is allocated, some positions may overlap other positions. Thus, for each position, we can add zero or one node. In contrast to the previously discussed layout containers, we can define the positions and nodes for a *BorderPane* in any order. **Figure 17.17** shows the five possible positions in a *BorderPane* layout.

Figure 17.17 The Five Positions of the *BorderPane* Layout

Description

To illustrate the *BorderPane* layout, we will write an application that animates a Sprite defined in **Example 17.34**. We've made some changes to the *Sprite* class. Because we want to animate the Sprite, we define the class as extending the *Canvas* class. The default constructor (lines 15–22) calls the constructor of the *Canvas* class, setting the width and height of the canvas to be the width and height of the Sprite. The overloaded constructor is similar (lines 24–35), except that it sets the canvas size to the Sprite's width and height multiplied by the validated *scale*. **Figure 17.18** shows the window when the *AnimationSampler* application begins, and **Example 17.35** shows the FXML file we use to define the layout and nodes.

Example 17.34 The *Sprite.java* File

```java
/* Sprite class
   Anderson, Franceschi
*/
import javafx.scene.canvas.*;
import javafx.scene.paint.*;

public class Sprite extends Canvas
{

  private static final int WIDTH = 95, HEIGHT = 138;
  private int sX;
  private int sY;
  private double scale;

  /** default constructor
  *  sX = sY = 0; scale is set to 1
  */
  public Sprite( )
  {
    super( WIDTH, HEIGHT );
    scale = 1;
  }

  /* overloaded constructor
  *  accepts values for starting x and y coordinates
  *  and scale
  */
  public Sprite( int sX, int sY, double scale )
  {
    super( );
    setCoordinates( sX, sY );
    this.scale = ( scale > 0.0 ? scale : 1 );
    setWidth( this.scale * WIDTH );
    setHeight( this.scale * HEIGHT );
  }

  /* setCoordinates
  * accepts new values for starting x and y;
  * returns a reference to this object
  */
  public Sprite setCoordinates( int sX, int sY )
  {
    this.sX = sX;
    this.sY = sY;
    return this;
  }

  /* mutator for scale
  *  returns a reference to this object
  */
  public Sprite setScale( double scale )
  {
    this.scale = ( scale > 0.0 ? scale : this.scale );
    return this;
  }

  /* draw method
  *  draws Sprite at current sX and sY
  *  multiplying lengths by scale
  *  accepts the GraphicsContext of the canvas
  */
  public void draw( GraphicsContext gc )
  {
    gc.setFill( Color.CORAL ); // body
    gc.fillOval( sX, sY + 15 * scale, 90 * scale, 120 * scale );
    gc.setFill( Color.DARKGOLDENROD ); // hat
    gc.fillRect( sX + 23 * scale, sY, 45 * scale, 22 * scale );
    gc.setStroke( Color.DARKGOLDENROD ); // hat brim
    gc.setLineWidth( 3 );
    gc.strokeLine( sX, sY + 23 * scale,
                   sX + 90 * scale, sY + 23 * scale );
    gc.setFill( Color.CHOCOLATE ); // eye
    gc.fillOval( sX + 60 * scale, sY + 45 * scale,
                 18 * scale, 12 * scale );
    gc.setFill( Color.DARKSALMON ); // feet
    gc.setLineWidth( 1 );
    gc.fillOval( sX + 45 * scale, sY + 125 * scale,
                 45 * scale, 12 * scale );
    gc.strokeOval( sX + 45 * scale, sY + 125 * scale,
                   45 * scale, 12 * scale );
    gc.fillOval( sX + 27 * scale, sY + 127 * scale,
                 45 * scale, 12 * scale );
    gc.strokeOval( sX + 27 * scale, sY + 127 * scale,
                   45 * scale, 12 * scale );
  }
}
```

Description

Figure 17.18 The *Animation Sampler* Window

Description

Example 17.35 The *fxml_animation.fxml* File

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <?import javafx.scene.control.*?>
 3 <?import javafx.scene.layout.*?>
 4
 5 <BorderPane xmlns:fx="http://javafx.com/fxml"
 6       fx:controller="AnimationController">
 7  <center>
 8   <VBox fx:id="canvasContainer" alignment="center"
 9      style="-fx-border-color:navy;-fx-border-width:1"/>
10  </center>
11  <left>
12   <VBox spacing="10" alignment="center"
13      style="-fx-border-color:navy;-fx-border-width:1;-fx-padding:8">
14
15     <Button text="Fade" fx:id="fadeButton" />
16     <Button text="Rotate" fx:id="rotateButton" />
17     <Button text="Scale" fx:id="scaleButton" />
18     <Button text="Combine Animations" fx:id="combineButton" />
19     <Button text="Start Translate" fx:id="startTranslateButton" />
20     <Button text="Stop Translate" fx:id="stopTranslateButton" />
21   </VBox>
22  </left>
23 </BorderPane>
```

Description

In Example 17.35, we define the top-level layout as a *BorderPane* (lines 5–6 and 23). We specify the *BorderPane* positions using FXML elements. In the *center* position, we place an empty *VBox* (lines 7–10). In the controller's *initialize* method, we will add the Sprite to this *VBox* and draw it. In the *left* position, we place the *VBox* containing the buttons (lines 11–22). Because at most one node can be placed into each position, we place multiple nodes into the *VBox* layout and place the *VBox* layout as the single node in the *left* position. We do not define any nodes for the *top*, *right*, or *bottom* positions. Thus, the *left* position is sized to fit the buttons, and the *center* position expands to fill the window. We define navy borders around the *VBoxes* (lines 9 and 13) so we can see their relative sizes and placement.

JavaFX supports two kinds of animations: transitions and timeline animations. Animations vary the properties of nodes, such as the opacity, position, or size, over a specified time period. In timeline animations, we can define custom variations of properties over time. In this chapter, we concentrate on predefined transitions.

Transitions use a technique called **tweening**, which gives the illusion of a property varying “between” two states by creating and displaying a series of intermediate frames over time. We can define individual transitions or combine multiple transitions to be played sequentially or in parallel.

JavaFX’s *abstract Transition* class inherits from the *abstract Animation* class. Each type of transition is a concrete subclass of the *Transition* class. All these classes are in the *javafx.animation* package.

To create an animation, we instantiate an object of the appropriate *Transition* subclass, tie the animation to a node, and specify the beginning and ending states of a property and the time period over which to vary the property. When ready, we play the transition. JavaFX automatically handles the variation of the property over time by creating intermediate frames. By default, the property varies from the start state to the end state at a constant rate.

In our *AnimationSampler* application, we demonstrate several kinds of transitions:

- *FadeTransition*, which varies the opacity of a node so that it fades in or out
- *RotateTransition*, which rotates a node a specified number of degrees
- *ScaleTransition*, which varies the size of a node
- *TranslateTransition*, which moves a node by varying its *x* or *y* coordinate, or both
- *SequentialTransition*, which plays a series of transitions one after the other
- *ParallelTransition*, which plays multiple transitions simultaneously

Some methods that all transitions inherit from the *Animation* class are shown in **Table 17.31**. The *setCycleCount* method allows us to specify how many times to run the animation. If we set the *autoReverse* property to *true*, the animation will play in reverse every other cycle. When we call the *play* method, the JavaFX platform manages the playing of the transition. We cannot alter the properties of the transition while it is playing. The only method that affects a running animation is *stop*. The *setOnFinished* method allows us to define an event handler to execute when the animation finishes. In addition, each transition has properties and methods specific to its animation.

Some Methods Common to *Animation* Subclasses

Package	**javafx.animation**
Common Methods of *Animation* Subclasses	
Return value	**Method name and argument list**
`void`	`play( )`
plays the animation from its current position	
`void`	`setAutoReverse( boolean value )`
if *value* is true, animation reverses on alternate cycles; if false, animation repeats from start position on each cycle	
`void`	`setCycleCount( int value )`
sets the animation to execute *value* times; if set to Animation.INDEFINITE, animation will run until stopped	
`void`	`setOnFinished( EventHandler<ActionEvent> handler )`
registers event handler's code to run when the animation finishes	
`void`	`stop( )`
stops the animation, if playing	

Description

We manage all the animations for our application in the controller, shown in **Example 17.36**. In the *initialize* method, we create our Sprite, which is a *Canvas*, and add it to the *VBox* we placed in the *center* position of the *BorderPane* (lines 34–37).

Example 17.36 *AnimationController.java*

```java
  1 /*  Animation Controller
  2     Anderson, Franceschi
  3 */
  4 import java.net.URL;
  5 
  6 import javafx.animation.*;
  7 import javafx.event.*;
  8 import javafx.fxml.*;
  9 import javafx.scene.canvas.*;
 10 import javafx.scene.control.*;
 11 import javafx.scene.layout.*;
 12 import javafx.scene.media.AudioClip;
 13 import javafx.util.Duration;
 14 
 15 public class AnimationController
 16 {
 17   @FXML private Button fadeButton, rotateButton,
 18                        scaleButton, combineButton,
 19                        startTranslateButton, stopTranslateButton;
 20   @FXML private VBox canvasContainer;
 21 
 22   private FadeTransition ftFade;
 23   private RotateTransition rtHalf, rtFull;
 24   private TranslateTransition ttStraight, ttDiagonal;
 25   private ScaleTransition stLarger, stSmaller, stSqueeze;
 26   private SequentialTransition sequential;
 27   private ParallelTransition parallel;
 28 
 29   private AudioClip sound;
 30 
 31   @FXML public void initialize( )
 32   {
 33     // Sprite extends Canvas
 34     Sprite sprite = new Sprite( 0, 0, 1 );
 35     GraphicsContext gc = sprite.getGraphicsContext2D( );
 36     sprite.draw( gc );
 37     canvasContainer.getChildren( ).add( sprite );
 38 
 39     // fade for 3 seconds, then reverse
 40     ftFade = new FadeTransition( Duration.seconds( 3 ), sprite );
 41     ftFade.setFromValue( 1.0 );
 42     ftFade.setToValue( 0.1 );
 43     ftFade.setCycleCount( 2 );
 44     ftFade.setAutoReverse( true );
 45     fadeButton.setOnAction( event -> runTransition( ftFade ) );
 46 
 47     // sound effect
 48     URL resource = getClass( ).getResource( "whoosh.m4a" );
 49     sound = new AudioClip( resource.toString( ) );
 50     sound.setCycleCount( 2 );
 51 
 52     // rotate 180 degrees and back
 53     rtHalf = new RotateTransition( Duration.millis( 3000 ), sprite );
 54     rtHalf.setFromAngle( 0 );
 55     rtHalf.setToAngle( 180 );
 56     rtHalf.setCycleCount( 2 );
 57     rtHalf.setAutoReverse( true );
 58     rotateButton.setOnAction( event ->
 59                               {
 60                                 runTransition( rtHalf );
 61                                 sound.play( );
 62                               }
 63                             );
 64 
 65     // reduce size to 1/2, return to start size
 66     stSmaller = new ScaleTransition( Duration.millis( 1500 ), sprite );
 67     stSmaller.setToX( 0.5f );
 68     stSmaller.setToY( 0.5f );
 69     stSmaller.setCycleCount( 2 );
 70     stSmaller.setAutoReverse( true );
 71 
 72     // reduce x size to 1/2, return to start size
 73     stSqueeze = new ScaleTransition( Duration.millis( 1500 ), sprite );
 74     stSqueeze.setToX( 0.5f );
 75     stSqueeze.setCycleCount( 2 );
 76     stSqueeze.setAutoReverse( true );
 77 
 78     sequential = new SequentialTransition( stSmaller, stSqueeze );
 79     scaleButton.setOnAction( event -> runTransition( sequential ) );
 80 
 81     // move right 80 pixels, then return to start
 82     ttStraight = new TranslateTransition( Duration.seconds( 3 ), sprite );
 83     ttStraight.setFromX( 0 );
 84     ttStraight.setToX( 80 );
 85     ttStraight.setCycleCount( 2 );
 86     ttStraight.setAutoReverse( true );
 87 
 88     // rotate 360 degrees
 89     rtFull = new RotateTransition( Duration.millis( 3000 ), sprite );
 90     rtFull.setByAngle( 360 );
 91 
 92     // increase size by half, return to start size
 93     stLarger = new ScaleTransition( Duration.millis( 1500 ), sprite );
 94     stLarger.setToX( 1.5f );
 95     stLarger.setToY( 1.5f );
 96     stLarger.setCycleCount( 2 );
 97     stLarger.setAutoReverse( true );
 98 
 99     parallel = new ParallelTransition( ttStraight, rtFull, stLarger );
100     combineButton.setOnAction( event -> runTransition( parallel ) );
101 
102     // move diagonally up 100 pixels, then return to start
103     ttDiagonal = new TranslateTransition( Duration.millis( 1500 ), sprite );
104     ttDiagonal.setFromX( 0 );
105     ttDiagonal.setToX( 100 );
106     ttDiagonal.setFromY( 0 );
107     ttDiagonal.setToY( -100 );
108     ttDiagonal.setCycleCount( Animation.INDEFINITE );
109     ttDiagonal.setAutoReverse( true );
110     stopTranslateButton.setDisable( true );
111     startTranslateButton.setOnAction( event ->
112                  {
113                     stopTranslateButton.setDisable( false );
114                     runTransition( ttDiagonal );
115                   } );
116 
117     stopTranslateButton.setOnAction( event ->
118                  {
119                     stopTranslateButton.setDisable( true );
120                     ttDiagonal.stop( );
121                     disableButtons( false );
122                   } );
123   }
124 
125   /**
126   *   runTransition
127   *   @param t, the transition to run
128   *   disables buttons, sets animation finished event handler
129   *   plays the animation
130   */
131   public void runTransition( Transition t )
132   {
133      disableButtons( true );
134      t.setOnFinished( event -> disableButtons( false ) );
135      t.play( );
136   }
137 
138   /** disableButtons
139   *   @param mode   true to disable, false to enable
140   */
141   public void disableButtons( boolean mode )
142   {
143      fadeButton.setDisable( mode );
144      startTranslateButton.setDisable( mode );
145      rotateButton.setDisable( mode );
146      scaleButton.setDisable( mode );
147      combineButton.setDisable( mode );
148   }
149 }
```

Description

In the *initialize* method, we define the animations, but we play them only in response to the user pressing a button. Our first transition is a *FadeTransition* that will fade the Sprite out, then in. A constructor for this transition and several useful methods are shown in **Table 17.32**.

A Constructor and Methods of the *FadeTransition* Class

Package	**javafx.animation**
A *FadeTransition* Constructor	
`FadeTransition( Duration time, Node node )`	
creates a *FadeTransition* that will animate *node* for a duration of *time*	
Selected Methods of the *FadeTransition* Class	
Return value	**Method name and argument list**
`void`	`setFromValue( double opacity )`
sets the beginning opacity of the animation; the *opacity* can range from 1.0 (fully opaque) to 0.0 (fully transparent)	
`void`	`setToValue( double opacity )`
sets the ending opacity of the animation; the *opacity* can range from 1.0 (fully opaque) to 0.0 (fully transparent)	

Description

At line 40, we instantiate our *FadeTransition* object. The first argument to the constructor is the length of time the animation should run, specified as a *Duration* object. The *Duration* class (in the *java.util* package) supplies several *static* factory methods to create *Duration* objects in units of milliseconds, seconds, minutes, or hours. These methods are shown in **Table 17.33**. Here we use the *static* factory method *seconds* to specify the length of the animation to be 3 seconds. The second argument of the constructor is the node we are animating, which is the Sprite we created.

Several *static* Factory Methods of the *Duration* Class

Package	**java.util**
Selected *static* Factory Methods of the *Duration* Class	
Return value	**Method name and argument list**
`Duration`	`millis( double ms )`
creates a *Duration* object for *ms* milliseconds	
`Duration`	`seconds( double secs )`
creates a *Duration* object for *secs* seconds	
`Duration`	`minutes( double mins )`
creates a *Duration* object for *mins* minutes	
`Duration`	`hours( double hrs )`
creates a *Duration* object for *hrs* hours	

Description

At lines 41–44, we define the properties of the transition. We set the opacity to vary from fully opaque (1.0) to almost transparent (0.1). We then set the cycle count to 2 so that the animation will repeat once. And we set *autoReverse* to *true* so that on the repeat cycle, the animation will reverse and the opacity will vary from almost transparent to fully visible. The result is that when this transition plays, the Sprite will appear to fade out for 3 seconds, then fade in for 3 seconds.

For running all our animations, we define a method, *runTransition* (lines 125–136), that accepts as a parameter the transition to play. This method calls the *disableButtons* method defined on lines 138–148 so that the user cannot attempt to start another transition while the current transition is running. The method then registers an *onFinished* event handler that re-enables the buttons when the transition ends, and then plays the transition.

Line 134 registers the *onFinished* event handler using a **lambda expression**.

In the examples in this chapter so far, we have defined event handlers in two ways: as *protected* methods that are referenced in the FXML file as the *onAction* property for a node, and as *private* inner classes in the controller. When an event handler is dedicated to only one control, a third option is to define the event handler as an **anonymous class**, that is, a class that we define inline without giving the class a name. Using this option, we can define and register the event handler at the same time. Let's explore this option with the event handler for the *onFinished* event handler. The only action for our event handler is to call the *disableButtons* method to enable the buttons when the animation finishes.

We can register our event handler using the *setOnFinished* method of the *Transition* class (inherited from the *Animation* class). To do this, we define a class that implements the *EventHandler<ActionEvent>* interface. To implement this class, we need to define one method, *handle*, shown in **Table 17.34**.

The *handle* Method of the *EventHandler<T extends Event>* Interface

Package	**javafx.event**
Return value	**Method name and argument list**
`void`	`handle( T event )`
provides code to run when *event* occurs on a registered control	

Description

In the *runTransition* method, we could register and define our event handler as an anonymous class that implements the

EventHandler interface using this code:

```
t.setOnFinished( new EventHandler<ActionEvent>( )
                  {
                     @Override
      public void handle( ActionEvent event )
      {
        disableButtons( false );
      }
   }
);
```

Description

Thus, the argument we send to the *setOnFinished* method is an object reference of the class that implements the *EventHandler<ActionEvent>* interface. We instantiate the object using *new* and define the class by providing the body of the *handle* method inline. Although this code is compact, misplacing a curly brace, parenthesis, or semicolon can generate errors that are difficult to debug. To write this code in a more readable form, we can use lambda expressions, introduced in Java Version 8.

Lambda expressions can be used only with **functional interfaces**, which are interfaces that have only one *abstract* method that the class implementing that interface must define. Thus, the *EventHandler* interface is a functional interface.

A lambda expression contains the following elements:

- A comma-separated list of parameters enclosed in parentheses. The data types of the parameters may be omitted. The parentheses may also be omitted if there is only one parameter.
- The arrow token, ->
- A method body, which can be a single expression or a block enclosed in curly braces
    - If the body of the method consists of a single expression, then the JVM evaluates the expression and returns its value. As an alternative, we can use a *return* statement, but that requires curly braces.

Given this syntax for lambda expressions, we can replace the code above with this single statement:

```
t.setOnFinished( event -> disableButtons( false ) );
```

In fact, this is line 134 in Example 17.36. When the animation finishes, we execute the *disableButtons* method to re-enable the buttons.

We have another opportunity to use lambda expressions in our button handlers. For example, we want the *FadeTransition* to play when the user presses the “Fade” button. This is the only action we want to register for the “Fade” button, and the *setOnAction* method takes an *EventHandler* object, so we are also able to set up an event handler for the button using a lambda expression. Following the syntax explained above, creating and registering the event handler for the “Fade” button is accomplished on line 45. The event handler calls the *runTransition* method passing the name of our *FadeTransition (ftFade)* as an argument.

We gain several advantages by using lambda expressions. The code is even more compact than defining an anonymous class; the code is more readable; and, by defining a separate event handler for each button, we avoid calling the *getSource* method from the event handler to determine which button the user clicked.

Our next animation is a *RotateTransition* that rotates the Sprite 180 degrees clockwise, then reverses to its original orientation while playing a sound effect.

The *AudioClip* class, in the *javafx.scene.media* package, is useful for playing short sounds, such as sound effects. When an *AudioClip* object is instantiated, the sound file is loaded entirely into memory. Thus, this class is intended for playing relatively short sounds. For longer sounds, we would use the *Media* class. You can read more about the *Media* class on Oracle’s Java website.

Table 17.35 shows the constructor and the *play* and *stop* methods of the *AudioClip* class. The constructor accepts a *String* representing a URL for the sound. In this case, we use a locally stored sound, although we could load a sound from the Internet.

A Constructor and Selected Methods of the *AudioClip* Class

Package	**javafx.scene.media**
***AudioClip* Constructor**	
`AudioClip( String source )`	
loads an audio clip from the URL represented by *source*	
Selected Methods of the *AudioClip* Class	
Return value	**Method name and argument list**
`void`	`setCycleCount( int count )`
count is the number of times to play the sound	
`void`	`play( )`
plays the sound	
`void`	`stop( )`
stops the sound, if playing	

Description

In addition to the methods shown in Table 17.35, the *AudioClip* class also provides methods for setting the rate and volume at which the sound is played and other properties. Like animations, we need to set all the properties of the sound before calling the *play* method. Once a sound is playing, the only method that affects it is *stop*.

At line 48, we instantiate a *URL* object for our sound file, *whoosh.m4a*. On line 49, we instantiate the *AudioClip* object. Because the *AudioClip* constructor accepts a *String* argument, we need to call the *toString* method on the *URL* object. We set the cycle count to 2 at line 50 so that the sound will play twice.

Now we're ready to set up the rotation animation. A constructor for the *RotateTransition* class and several useful methods are shown in **Table 17.36**. When calling those methods, we specify angle values in degrees.

A Constructor and Methods of the *RotateTransition* Class

Package	**javafx.animation**
A *RotateTransition* Constructor	
RotateTransition(Duration time, Node node)	
creates a *RotateTransition* that will animate *node* for a duration of *time*	
Selected Methods of the *RotateTransition* Class	
Return value	**Method name and argument list**
`void`	`setFromAngle( double startAngle )`
sets the angle from which to begin the rotation	
`void`	`setToAngle( double endAngle )`
sets the angle at which to end the rotation	
`void`	`setByAngle( double angle )`
sets the angle relative to the start at which to stop	

Description

On lines 52–57, we instantiate the transition and set the rotation for 180 degrees, to repeat in reverse.

When the user presses the *Rotate* button, we want both to start the animation and to play the sound. That means that our event handler (lines 58–63) consists of more than one statement, so, using a lambda expression, we need to place the two statements inside curly braces. Otherwise, the syntax is the same as for the fade animation.

Our next animation is actually two animations that we play sequentially. Using *ScaleTransitions*, we decrease, then "squeeze" the size of the Sprite. A constructor for the

ScaleTransition class and several useful methods are shown in **Table 17.37**. When calling those methods, we specify relative, not absolute, values.

A Constructor and Methods of the *ScaleTransition* Class

Package	**javafx.animation**
A *ScaleTransition* Constructor	
`ScaleTransition( Duration time, Node node )`	
creates a *ScaleTransition* that will animate *node* for a duration of *time*	
Selected Methods of the *ScaleTransition* Class	
Return value	**Method name and argument list**
`void`	`setToX( double value )`
sets the x scale at which to stop	
`void`	`setToY( double value )`
sets the y scale at which to stop	

Description

At lines 65–70, we instantiate a *ScaleTransition* and decrease the size of the Sprite by setting its *x* and *y* scales to 50%. The *x* scale determines how large the node is horizontally, and the *y* scale determines how large the node is vertically. At lines 72–76, we create another *ScaleTransition* that decreases only the *x* scale by 50%. Because we are not changing the *y* scale, when this animation plays, the Sprite appears to be squeezed horizontally.

We want to play these two animations one after the other, so we create a *SequentialTransition* and add both animations as children (line 78). The *SequentialTransition* constructor shown in **Table 17.38** accepts children as a *varargs* argument, so we can add as many animations as desired. When the user presses the “Scale” button, we start the *SequentialTransition* (line 79), which plays the *stSmaller* animation to completion, then automatically plays the *stSqueeze* animation.

A Constructor of the *SequentialTransition* Class

Package	**javafx.animation**
`SequentialTransition( Animation… children )`	
creates a *SequentialTransition* that will play all *children* animations in sequence	

Description

Another way to combine animations is to define a *ParallelTransition*, which plays multiple animations simultaneously. For this, we create three animations: moving the Sprite horizontally, rotating the Sprite, and changing its scale.

To move the Sprite horizontally, we define a *TranslateTransition*, which allows us to change the *x* or *y* coordinate of a node. A constructor for the *TranslateTransition* class and several useful methods are shown in **Table 17.39**.

A Constructor and Methods of the *TranslateTransition* Class

Package	**javafx.animation**
A *TranslateTransition* Constructor	
`TranslateTransition( Duration time, Node node )`	
creates a *TranslateTransition* that will animate *node* for a duration of	

time

Selected Methods of the *TranslateTransition* Class

Return value	Method name and argument list
`void`	`setFromX( double value )`
sets the starting x value	
`void`	`setToX( double value )`
sets the ending x value	
`void`	`setFromY( double value )`
sets the starting y value	
`void`	`setToY( double value )`
sets the ending y value	

Description

We define the *TranslateTransition* at lines 81–86. We move the Sprite right by increasing the *x* coordinate by 80 pixels. Then, because we have two cycles and we have set *autoReverse* to *true*, the Sprite returns to its starting position.

We define the second animation in our set to rotate the Sprite 360 degrees (lines 88–90). We do not set a cycle count or *autoReverse*, so these properties default to one cycle and no *autoReverse*. As a result, the animation will run once.

Our final animation in this set scales the Sprite to 1.5 times and then back to the original size (lines 92–97).

Using the constructor in **Table 17.40**, we now create a *ParallelTransition*, adding our three animations (line 99). When the user presses the “Combine Animations” button, we set the listener to play the parallel animation (line 100), which starts all three animations simultaneously. Notice that not all animations in the set need to have the same duration. For the first 1.5 seconds, the Sprite rotates 180 degrees, moves right 40 pixels, and scales to 1.5 times its size. For the next 1.5 seconds, the Sprite completes its move right, rotates the remaining 180 degrees, and scales back to its original size. For the last 3 seconds, the Sprite moves left to its starting position.

A Constructor of the *ParallelTransition* Class

Package	**javafx.animation**

`ParallelTransition( Animation… children )`

creates a *ParallelTransition* that will play all *children* animations simultaneously

---

Description

For the final animation, we define a *TranslateTransition* that alters both the *x* and *y* values so that the Sprite moves diagonally up to the right (lines 104–111). Note that we set a cycle count of *Animation.INDEFINITE*, which specifies that the animation should keep running until the *stop* method is called. When the user presses the “Start Translate” button, we start the animation, and when the user presses the “Stop Translate” button, we stop the animation. Because the “Stop Translate” button should be enabled only while the animation is running, we set its initial state as disabled (line 110). In the event handler for the “Start Translate” button, we enable the “Stop Translate” button, then start the animation (lines 111–115). In the event handler for the “Stop Translate” button, we disable the “Stop Translate” button, call the *stop* method, then re-enable the rest of the buttons (lines 117–122).

As an animation plays, a **play head** keeps track of the current frame. When we call the *stop* method, the animation stops playing, and its play head is reset to the beginning of the animation. The node, however, remains in the state at which it was stopped. When we again press the “Start Translate” button, we see that because the play head has been reset, the Sprite returns to the start position before beginning the animation.

Example 17.37 shows the *AnimationSampler* code that launches the application.

Example 17.37 *AnimationSampler.java*

```
/*  Animation Sampler
    demonstrates various animations
    Anderson, Franceschi
*/

import java.net.URL;
import javafx.application.*;
import javafx.fxml.*;
import javafx.scene.layout.*;
import javafx.scene.*;
import javafx.stage.*;

public class AnimationSampler extends Application
{
  @Override
  public void start( Stage stage ) // throws Exception
  {
    try
    {
      URL url =
         getClass( ).getResource( "fxml_animation.fxml" );
      BorderPane root = FXMLLoader.load( url );
      Scene scene = new Scene( root, 800, 600 );
      stage.setTitle( "Animation Sampler" );
      stage.setScene( scene );
      stage.show( );
    }
    catch ( Exception e )
    {
      e.printStackTrace( );
    }
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

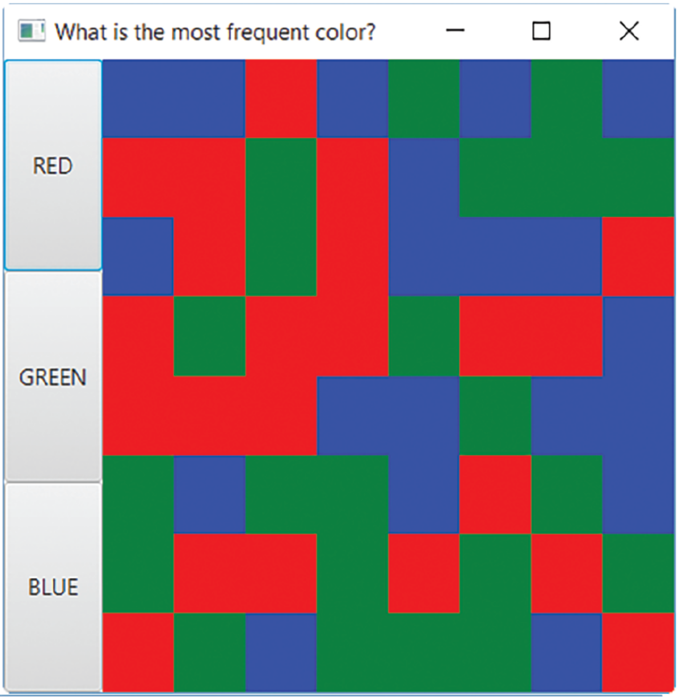
Description

CODE IN ACTION

Within the online resources, you will find a movie with a step-by-step illustration of working with layout containers and animation. Click on the link to start the movie.

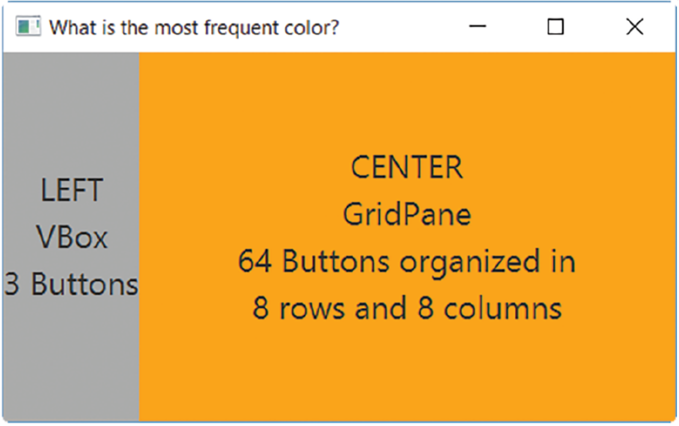
© Hemera Technologies/Photos.com/Thinkstock

(A)



What is the most frequent color?

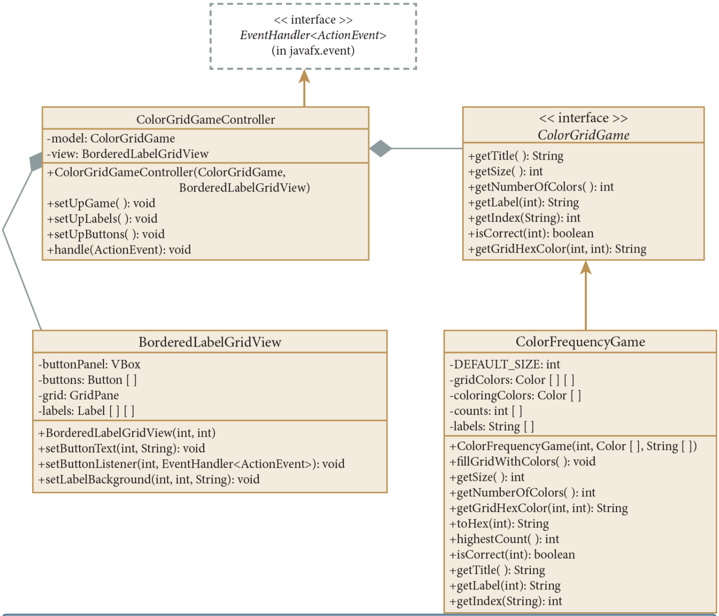
(B)



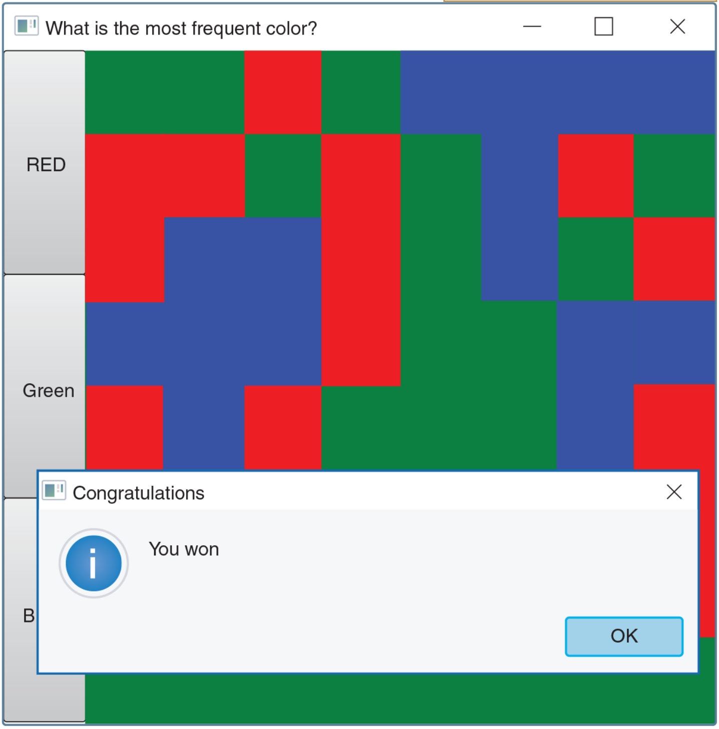
What is the most frequent color?

LEFT
VBox
3 Buttons

CENTER
GridPane
64 Buttons organized in
8 rows and 8 columns



<< interface >>
EventHandler<ActionEvent>
(in javafx.event)
ColorGridGameController
-model: ColorGridGame
-view: BorderedLabelGridView
+ColorGridGameController(ColorGridGame, BorderedLabelGridView)
+setUpGame( ): void
+setUpLabels( ): void
+setUpButtons( ): void
+handle(ActionEvent): void
<< interface >>
ColorGridGame
+getTitle( ): String
+getSize( ): int
+getNumberOfColors( ): int
+getLabel(int): String
+getIndex(String): int
+isCorrect(int): boolean
+getGridHexColor(int, int): String
BorderedLabelGridView
-buttonPanel: VBox
-buttons: Button [ ]
-grid: GridPane
-labels: Label [ ] [ ]
+BorderedLabelGridView(int, int)
+setButtonText(int, String): void
+setButtonListener(int, EventHandler<ActionEvent>): void
+setLabelBackground(int, int, String): void
ColorFrequencyGame
-DEFAULT_SIZE: int
-gridColors: Color [ ] [ ]
-coloringColors: Color [ ]
-counts: int [ ]
-labels: String [ ]
+ColorFrequencyGame(int, Color [ ], String [ ])
+fillGridWithColors( ): void
+getSize( ): int
+getNumberOfColors( ): int
+getGridHexColor(int, int): String
+toHex(int): String
+highestCount( ): int
+isCorrect(int): boolean
+getTitle( ): String
+getLabel(int): String
+getIndex(String): int


What is the most frequent color?

Congratulations

You won

RED
Green
B
OK

17.13 Nesting Components

Components can be nested. Indeed, because the layout containers are subclasses of *Parent*, itself a subclass of *Node*, layout containers are both containers and components. As such, they can contain other layout containers, which in turn can contain components. We can use this feature to achieve more precise layouts.

When nesting components, we usually place several components into a layout container and place that layout container into another layout container. Each layout container can be different so that components can be arranged in many ways. We can even have multiple levels of nesting, as needed.

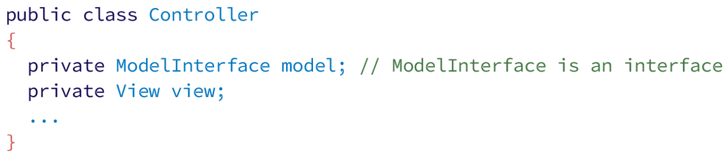
We illustrate nesting of components in an example where, using three colors (red, green, and blue), we randomly generate an 8-by-8 grid of colors. The user tries to guess the most frequently occurring color by clicking on one of the buttons.

Figure 17.19A shows our window when the application begins, and **Figure 17.19B** shows the underlying layout of the window. We use a *BorderPan*e overall, with a *VBox* containing the three buttons on the left side of the *BorderPane*, and a *GridPane* with eight rows and eight columns containing the 64 colored labels in the center position of the *BorderPane*; thus, the two positions we are using, left and center, adjust to fill the space.

Figure 17.19 (A) The Color Frequency Game (B) The Underlying Nested Layout

Description

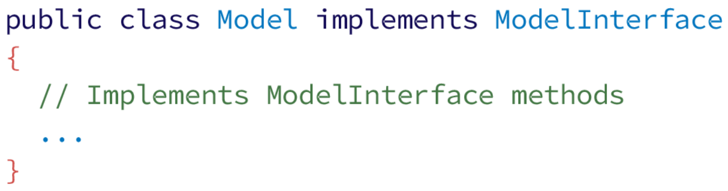
Again, we use the MVC architecture and design our classes for reusability. This time, we also code the Controller so that it is reusable under certain conditions. Inside the *Controller* class, instead of using a class for the Model, we use an interface. Thus, the Controller is now reusable, along with the same View, with any Model that implements that interface. Note that we could also make the View an interface, but in order to keep this example relatively simple, we do not. To achieve reusability for the Controller, we use the following pattern for the *Controller* class:



```
public class Controller
{
  private ModelInterface model; // ModelInterface is an interface
  private View view;
  ...
}
```

Description

The *ModelInterface* specifies several methods to be implemented that the Controller will call to manage the game. The *Model* class uses this pattern:



```
public class Model implements ModelInterface
{
  // Implements ModelInterface methods
  ...
}
```

Description

We want the *Controlle*r class to be reusable with any class whose functionality is based on the user clicking buttons on the left panel with a grid of labels in the center position. We could use this class in an application where the user tries to guess the most frequent color, or the least frequent color, or with a different color system (e.g., HSL—Hue, Saturation, and Lightness).

Thus, we design an interface to support this type of application that has buttons on the left and a grid of labels in the center. **Example 17.38** shows the *ColorGridGame* interface. It specifies methods to retrieve the title for the grid, its size, its number of colors, the label of a color, and the color index of a label. (It is implied that the colors used in the grid of colors will come from an array of colors, and that there will be an array of *Strings* [labels] that parallels that array of colors.) It also specifies methods to test if an answer is the correct answer, as well as to access each color in the grid. We call these methods from the Controller class with the Model instance variable in order to manage the game. The UML diagram for this example is shown in **Figure 17.20**.

Example 17.38 The *ColorGridGame* Interface

```
/* ColorGridGame interface
*  Anderson, Franceschi
*/

import javafx.scene.paint.Color;

public interface ColorGridGame
{
 public String getTitle( );
 public int getSize( );
 public int getNumberOfColors( );
 public String getLabel( int index );
 public int getIndex( String label );
 public boolean isCorrect( int index );
 public String getGridHexColor( int row, int col );
}
```

Description

Figure 17.20 The UML Diagram for the Color Frequency Game Application

Description

We now code our Model for the color frequency game. In this application, we generate colors for our grid of labels. **Example 17.39** shows the *ColorFrequencyGame* class, which *implements ColorGridGame* (line 8). At lines 10 through 12, we declare a *char* array storing hexadecimal digits. We use this array to convert an integer to its hexadecimal equivalent *String*. Its *gridColors* instance variable (line 15) stores a two-dimensional array, or grid, of colors. The *coloringColors* and *counts* arrays (lines 16–17) are parallel arrays: *coloringColors* stores the colors used for coloring the grid, and *counts* stores the frequency of each color. The instance variable *labels* (line 19) is an array of *Strings* that is meant to be labels for the buttons. The *labels* array also parallels the *coloringColors* array.

Example 17.39 The *ColorFrequencyGame* Class

```java
/** ColorFrequencyGame class
    Anderson, Franceschi
*/

import java.util.Random;
import javafx.scene.paint.Color;

public class ColorFrequencyGame implements ColorGridGame
{
 private static char [ ] hexDigits =
    { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
      'A', 'B', 'C', 'D', 'E', 'F' };

 private final int DEFAULT_SIZE = 8;
 private Color [ ][ ] gridColors;
 private Color [ ] coloringColors;
 private int [ ] counts; // color frequencies in gridColors

 private String [ ] labels;

 /** Constructor
 * @param size number of rows and columns in gridColors
 * @param colors array of colors
 * @param labels the starting labels
 */
 public ColorFrequencyGame( int size, Color [ ] colors,
                            String [ ] labels )
 {
  if ( size <= 0 )
   size = DEFAULT_SIZE;
  gridColors = new Color[size][size];
  coloringColors = colors;
  counts = new int[coloringColors.length];
  fillGridWithColors( );
  this.labels = labels;
 }

 /** fillGridWithColors method
 * randomly fills gridColors with colors from coloringColors
 */
 public void fillGridWithColors( )
 {
  Random random = new Random( );
  for ( int i = 0; i < gridColors.length; i++ )
  {
   for ( int j = 0; j < gridColors[i].length; j++ )
   {
    int colorIndex = random.nextInt( coloringColors.length );
    gridColors[i][j] = coloringColors[colorIndex];
    counts[colorIndex] += 1;
   }
  }
 }

 /** getSize method
 * @return length of gridColors
 */
 public int getSize( )
 {
  return gridColors.length;
 }

 /** getNumberOfColors method
 * @return length of coloringColors
 */
 public int getNumberOfColors( )
 {
  return coloringColors.length;
 }

 /** getGridHexColor method
 * @param row, an int, the row index
 * @param col, an int, the column index
 * @return a String, the hex equivalent of gridColor[row][col]
 */
 public String getGridHexColor( int row, int col )
 {
  String colorText = "#"
    + toHex( ( int ) ( 255 * gridColors[row][col].getRed( ) ) )
    + toHex( ( int ) ( 255 * gridColors[row][col].getGreen( ) ) )
    + toHex( ( int ) ( 255 * gridColors[row][col].getBlue( ) ) );
  return colorText;
 }

 /** toHex method
 * @return a String, the Hex equivalent of colorIntensity
 */
 public String toHex( int colorIntensity )
 {
  if ( colorIntensity > 255 )
    colorIntensity = 255;
  else if ( colorIntensity < 0 )
    colorIntensity = 0;

  int firstDigit = colorIntensity / 16;
  int secondDigit = colorIntensity % 16;
  return String.valueOf( hexDigits[firstDigit]) + hexDigits[secondDig
 }

 /** highestCount method
 * @return the highest color frequency in the grid
 */
 public int highestCount( )
 {
  int max = counts[0];
  for ( int i = 1; i < counts.length; i++ )
  {
   if ( counts[i] > max )
    max = counts[i];
  }
  return max;
 }

 /** isCorrect method
 * @param index, the index in the coloringColors array to check
 * @return true if the frequency of the color for index
 * is the highest, false otherwise
 */
 public boolean isCorrect( int index )
 {
  return counts[index] == highestCount( );
 }

 /** getTitle method
 * @return a String representing a title for this object
 */
 public String getTitle( )
 {
  return "What is the most frequent color?";
 }

 /** getLabel method
 * @param index, the index of the Color name
 * @return the color name
 */
 public String getLabel( int index )
 {
  return labels[index];
 }

 /** getIndex method
 * @param label, a color name in the labels array
 * @return index of the color name in the labels array
 */
 public int getIndex( String label )
 {
  for ( int i = 0; i < labels.length; i++ )
  {
   if ( labels[i].equals( label ) )
    return i;
  }
  return -1;
 }
}
```

Description

The constructor (lines 21–36) instantiates all three arrays and calls the *fillGridWithColors* method at line 34. The *fillGridWithColors* method (lines 38–53) fills the *colorGrid array* with colors randomly generated from the *coloringColors* array and keeps track of their respective frequency in the *counts* array (line 50).

The *getGridHexColor* method (lines 71–83) returns a hexadecimal representation of the color stored at a specified row and column within the array *gridColors*. It uses the utility method *toHex* (lines 85–98), which converts an integer whose value is between 0 and 255 to its hexadecimal equivalent *String*.

To determine whether the user has chosen the correct color, we provide the *isCorrect* method (lines 114–122), which takes as a parameter the index of the selected color. It returns *true* if the value in *counts* at that index is maximal, *false* otherwise. The *highestCount* method (lines 100–112) calculates and returns the maximum value in the *counts* array. Note that if multiple colors appear with the highest count of labels—essentially a tie—then picking any of those colors will result in a winning selection.

The *getTitle* method (lines 124–130) returns a *String* that can be used as the title of a GUI whose Model is the *ColorFrequencyGame* class. The *getLabel* method (lines 132–139) returns the element of the array *labels* at a specified index, sent as the parameter of the method; that is, the label is to be used with a color located at that index within the array *coloringColors*. The *getIndex* method (lines 141–153) is the opposite method: It returns the index within the array *labels* of a *String*, sent as the parameter of the method; the color corresponding to that label is the element in the array *coloringColors* at that index.

Example 17.40 shows the *BorderedLabelGridView* class. It is a layout container class that *extends BorderPane* (line 16). At its left position, it contains an array of buttons arranged vertically. At its center position, it contains a grid of labels.

Example 17.40 The *BorderedLabelGridView* Class

```java
/** BorderedLabelGridView class
* Reusable generic layout using a BorderPane's
* left and center positions.
* The left VBox is made up of a vertical array of buttons.
* The center position is made up of a grid of labels.
* Accessors are provided so that a Controller can access
* the array of buttons and the 2-dim array (grid) of labels.
* Anderson, Franceschi
*/

import javafx.event.*;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;

public class BorderedLabelGridView extends BorderPane
{
 private VBox buttonPanel; // left, holds array of buttons
 private Button [ ] buttons;
 private GridPane grid; // center, holds grid of labels
 private Label [ ][ ] labels; // grid of labels

 // numberOfButtons and gridSize must be greater than 0
 public BorderedLabelGridView( int numberOfButtons, int gridSize )
 {
  super( );
  grid = new GridPane( );

  // set up grid as gridSize by gridSize
  RowConstraints row = new RowConstraints( );
  row.setPercentHeight( 100.0 / gridSize );
  ColumnConstraints col = new ColumnConstraints( );
  col.setPercentWidth( 100.0 / gridSize );
  for ( int i = 0; i < gridSize; i++ )
    grid.getRowConstraints( ).add( row );
  for ( int j = 0; j < gridSize; j++ )
    grid.getColumnConstraints( ).addAll( col );

  labels = new Label[gridSize][gridSize];
  for ( int i = 0; i < labels.length; i++ )
  {
   for ( int j = 0; j < labels[i].length; j++ )
   {
    labels[i][j] = new Label( );
    // make label fill up available width and height
    labels[i][j].setMaxWidth( Double.MAX_VALUE );
    labels[i][j].setMaxHeight( Double.MAX_VALUE );
    grid.add( labels[i][j], j, i );
   }
  }

  buttonPanel = new VBox( );

  buttons = new Button[numberOfButtons];
  for ( int i = 0; i < buttons.length; i++ )
  {
   buttons[i] = new Button( );
   // make button fill up available width and height
   buttons[i].setMaxWidth( Double.MAX_VALUE );
   buttons[i].setMaxHeight( Double.MAX_VALUE );
   VBox.setVgrow( buttons[i], Priority.ALWAYS );
   buttonPanel.getChildren( ).add( buttons[i] );
  }

  setLeft( buttonPanel );
  setCenter( grid );
 }

 public void setButtonText( int row, String text )
 {
  buttons[row].setText( text );
 }

 public void setButtonListener( int row, EventHandler<ActionEvent> eh )
 {
  buttons[row].setOnAction( eh );
 }

 public void setLabelBackground( int row, int col, String hexColor )
 {
  labels[row][col].setStyle( "-fx-background-color: " + hexColor );
 }
}
```

Description

Thus, our instance variables are a *VBox* (line 18), which contains an array of *Buttons* (line 19), and a *GridPane* (line 20), which contains a two-dimensional array of *Labels* (line 21). We place the *VBox* in the left position and the *GridPane* in the center position. The other positions within the *BorderPane* are not used.

The constructor (lines 23–67) defines a GUI consisting of an array of buttons on the left, arranged vertically, and a grid of labels in the center, assumed to be a square. Its two parameters represent the number of buttons and the size of the grid. For simplicity, we are not validating these parameters, but both parameters must be greater than 0. The grid of labels is defined at lines 27 through 50 inside the *GridPane*. The array of buttons is defined at lines 52 through 63 inside the *VBox*. At lines 65 and 66, we add the *VBox* and the *GridPane* to this *BorderPane* in the left and center positions, respectively. The *BorderPane* class provides similar methods (*setCenter, setLeft, setRight, setTop,* and *setBottom*) for adding a component to each of its five positions.

We instantiate *grid*, the *GridPane* instance variable, at line 27. At lines 29 through 37, we define row and column constraints so that all the rows in the grid have the same height, all the columns have the same width, and the rows and columns take all the available space in the grid. At lines 39 through 50, we instantiate the labels and add them to the grid, making sure that each label fills the available width and height (lines 45–47).

We instantiate *buttonPanel*, the *VBox* instance variable, at line 52. At lines 54 through 63, we instantiate the buttons and add them to *buttonPanel*, also making sure that each label fills the available width and height (lines 58–61). It is important to set not only the maximum height of the buttons to a high value (line 60), but also their maximum width (line 59). If we do not, the buttons will have uneven width because their text contents are different. It is also important to call the *setVgrow* method (line 61), shown in **Table 17.41**, so that the buttons are expanded vertically in order to fill the available space within the *VBox* to which they belong. The *setVgrow* method sets the priority level for expanding the height of a node that is contained in the *VBox* container. The *setVgrow* method is *static*, so there is no reference to a specific *VBox* when we call this method;

thus, it is assumed that the node is or will be contained in a *VBox*. If we do not call *setVgrow*, the heights of the buttons are not expanded. *Priority* is an *enum* with three possible values: *ALWAYS*, *NEVER*, and *SOMETIMES*. The *ALWAYS* value ensures that the node grows in order to fill the available space in the *VBox* container. We specify the same priority level for all of the buttons so that they grow equally.

The *setVgrow* Method of the *VBox* Class

Package	**javafx.scene.layout**
Return value	**Method name and argument list**
`void`	`setVgrow( Node child, Priority value )`
static method that sets the vertical grow priority of *child*, assuming that *child* is contained in a *VBox*. If set, the *VBox* will allocate additional space for *child* if space is available.	

Description

Finally, we provide methods so that the text of each button and the background color of each label can be set from outside the class (lines 69–72 and 79–82). We also provide a method (lines 74–77) to set an *EventHandler* for each button in the array *buttons*. By calling these methods, the Controller can set up event handling and update this View.

Example 17.41 shows the *ColorGridGameController* class. At lines 13 and 14, we declare our two instance variables: (1) *model*, a *ColorGridGame*; and (2) *view*, a *BorderedLabelGridView*. These instance variables enable us to get user input from the View, call the appropriate methods of the Model, and update the View accordingly.

Example 17.41 The *ColorGridGameController* Class

```java
/** ColorGridGameController class
*   Anderson, Franceschi
*/

import javafx.event.*;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;

public class ColorGridGameController implements EventHandler<ActionEvent>
{
 private ColorGridGame model;
 private BorderedLabelGridView view;

 public ColorGridGameController( ColorGridGame model,
                                 BorderedLabelGridView view )
 {
  this.model = model;
  this.view = view;
  setUpGame( );
 }

 public void setUpGame( )
 {
  setUpLabels( );
  setUpButtons( );
 }

 public void setUpLabels( )
 {
  for ( int i = 0; i < model.getSize( ); i++ )
   for ( int j = 0; j < model.getSize( ); j++ )
    view.setLabelBackground( i, j, model.getGridHexColor( i, j ) );
 }

 public void setUpButtons( )
 {
  for ( int i = 0; i < model.getNumberOfColors( ); i++ )
  {
   view.setButtonText( i, model.getLabel( i ) );
   view.setButtonListener( i, this );
  }
 }

 public void handle( ActionEvent event )
 {
  Button button = ( Button ) event.getSource( );
  int index = model.getIndex( button.getText( ) );

  if ( model.isCorrect( index ) )
      showMessageDialog( "Congratulations", "You won" );
  else
      showMessageDialog( "Sorry", "You lost" );
  // disable buttons here
 }

 public void showMessageDialog( String title, String message )
 {
   Alert alert = new Alert (AlertType.INFORMATION );
   alert.setTitle( title );
   alert.setHeaderText( "" );
   alert.setHeaderText( message );
   alert.showAndWait( );
 }
}
```

Description

The constructor (lines 16–22) accepts two parameters that it assigns to *model* and *view*, and it calls the *setUpGame* method at line 21. The *setUpGame* method (lines 24–28) calls the *setUpLabels* and *setUpButtons* methods.

The *setUpLabels* method (lines 30–35) uses a nested loop to color the grid of labels in the View based on the grid of colors in the Model. At line 34, we set the background color of the current label by calling the View's *setLabelBackground* method, passing the row index, column index, and background color that we retrieve from the Model by calling the *getGridHexColor* method.

The *setUpButtons* method (lines 37–44) places a label on each button in the View based on the label data in the Model and sets up event handling using a single loop. At line 41, we set the text for the current button, passing its index and the label that we retrieve from the Model by calling the *getLabel* method. At line 42, we register *this* object (i.e., this controller, which implements *EventHandler*) on the current button.

We handle the events for the buttons in the *handle* method at lines 46 through 56. We first retrieve the button that originated the event by calling *getSource* at line 48. The *getSource* method returns an *Object*, so we need to type cast the return value to a *Button*. We then retrieve the index of that button's corresponding label by calling the Model's *getIndex* method, passing the text of the button. Next, we check whether the user won or lost by calling the Model's *isCorrect* method, passing that index. Depending on the result, we pop up a dialog box indicating whether the user has won or lost (lines 51–54). The *showMessageDialog* method (lines 58–65) is identical to the one in the puzzle game. After the game is won or lost, we should disable the buttons. This is left as an exercise.

Finally, the *PlayColorCount* class, shown in **Example 17.42**, includes the *main* and *start* methods to create a color count application. We declare an array of three *Colors* at line 16 and a parallel array of *Strings* at line 17, and pass these arrays to the constructor of the *ColorFrequencyGame* at lines 19 through 20. We then create the *BorderedLabelGridView* object *root* at lines 22 and 23. Using *game* and *root*, we create a *ColorGridGameController* at lines 25 and 26. At line 29, we retrieve and set the title of the window. Note that *gam*e "is a*"*

ColorGridGame object because *ColorFrequencyGame* inherits from *ColorGridGame*. **Figure 17.21** shows a run of this example after the user has clicked on the button “GREEN.” To play the game with more colors, we could simply add more colors and labels to the *colors* and *labels* array.

Example 17.42 The *PlayColorCount* Class

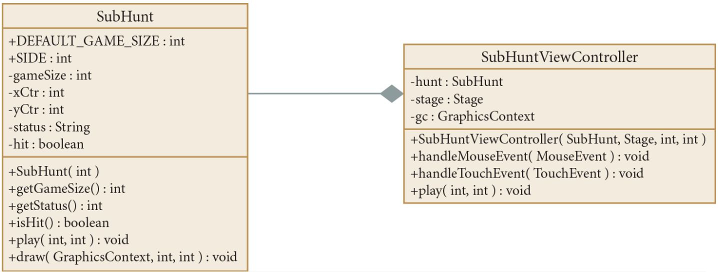
```java
/** PlayColorCount class
*   Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.scene.paint.Color;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class PlayColorCount extends Application
{

  @Override
  public void start( Stage stage )
  {
    Color [ ] colors = { Color.RED, Color.GREEN, Color.BLUE };
    String [ ] labels = { "RED", "GREEN", "BLUE" };

    ColorFrequencyGame game
      = new ColorFrequencyGame( 8, colors, labels );

    BorderedLabelGridView root
      = new BorderedLabelGridView( colors.length, game.getSize( ) );

    ColorGridGameController controller
      = new ColorGridGameController( game, root );

    Scene scene = new Scene( root, 450, 425 );
    stage.setTitle( game.getTitle( ) );
    stage.setScene( scene );
    stage.show( );
}

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

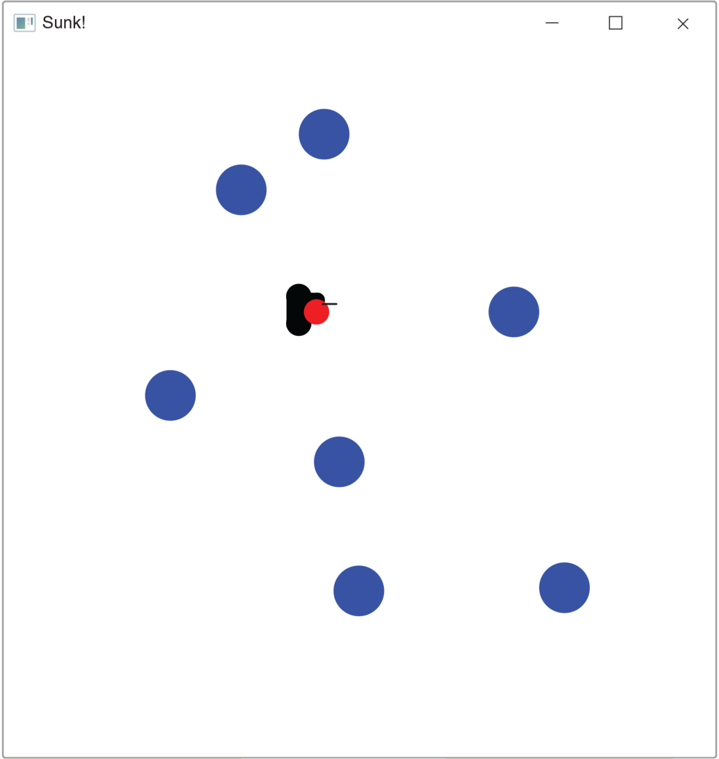
Description

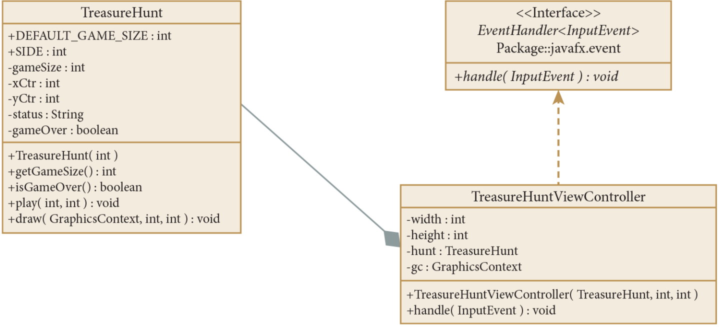
Figure 17.21 The User Clicked on the Button GREEN

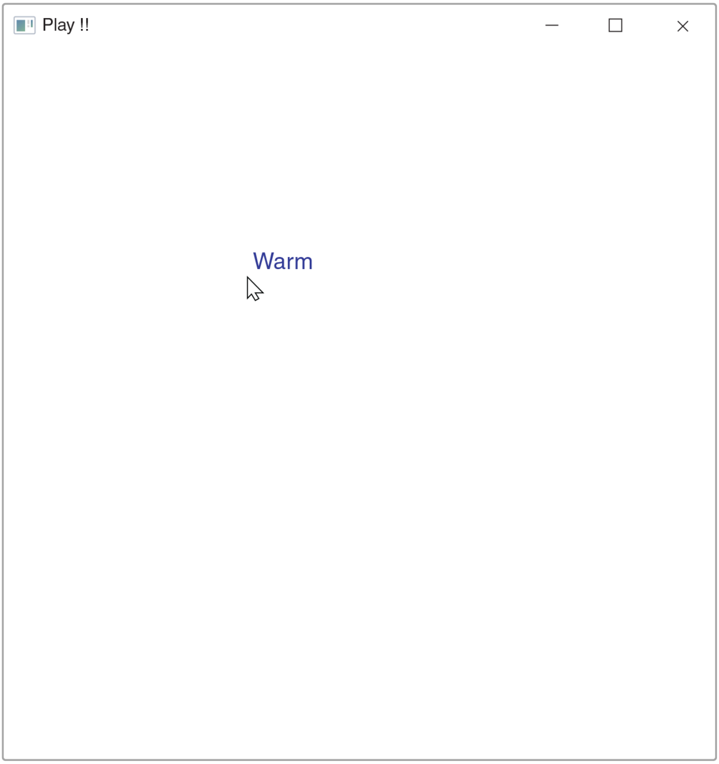
with these end-of-chapter questions

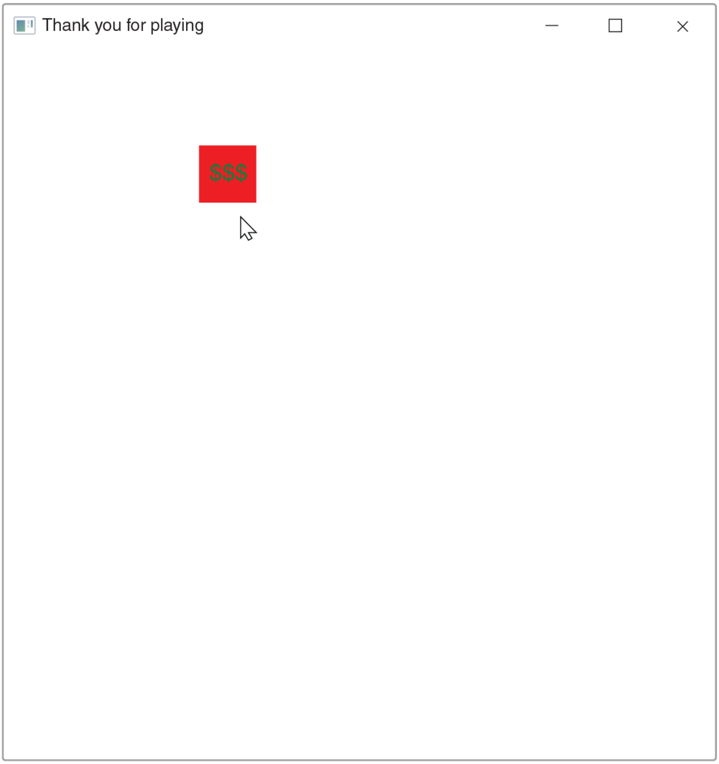
17.20.1	Multiple Choice Exercises Questions 10, 11, 12, 13, 14, 15, 16, 17, 18
17.20.2	Reading and Understanding Code Questions 30, 31, 32
17.20.3	Fill In the Code Questions 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49
17.20.4	Identifying Errors in Code Questions 54, 55, 56
17.20.5	Debugging Area—Using Messages from the Java Compiler and Java JVM Question 63
17.20.6	Write a Short Program Questions 67, 73, 74, 76, 78, 79, 80


SubHunt
+DEFAULT_GAME_SIZE : int
+SIDE : int
-gameSize : int
-xCtr : int
-yCtr : int
-status : String
-hit : boolean
+SubHunt(int)
+getGameSize() : int
+getStatus() : int
+isHit() : boolean
+play(int, int) : void
+draw(GraphicsContext, int, int) : void
SubHuntViewController
-hunt : SubHunt
-stage : Stage
-gc : GraphicsContext
+SubHuntViewController(SubHunt, Stage, int, int)
+handleMouseEvent(MouseEvent) : void
+handleTouchEvent(TouchEvent) : void
+play(int, int) : void


Sunk!


TreasureHunt
+DEFAULT_GAME_SIZE : int
+SIDE : int
-gameSize : int
-xCtr : int
-yCtr : int
-status : String
-gameOver : boolean
+TreasureHunt(int)
+getGameSize() : int
+isGameOver() : boolean
+play(int, int) : void
+draw(GraphicsContext, int, int) : void
<<Interface>>
EventHandler<InputEvent>
Package::javafx.event
+handle(InputEvent) : void
TreasureHuntViewController
-width : int
-height : int
-hunt : TreasureHunt
-gc : GraphicsContext
+TreasureHuntViewController(TreasureHunt, int, int)
+handle(InputEvent) : void


Play !!
Warm



17.14 Mouse and Touch Events

A truly interactive application allows the user to point and click using the mouse or to touch the screen. Any mouse activity (clicking, moving, or dragging) by the user generates a *MouseEvent*. Any touch activity by the user generates a *TouchEvent*. When any mouse or touch activity occurs, we will be interested in determining where it happened on the window. To determine the (*x, y*) coordinate of a mouse event, we can call two methods of the *MouseEvent* class, *getSceneX* and *getSceneY*, which are described in **Table 17.42**. To determine the (*x, y*) coordinate of a touch event, we can first call the *getTouchPoint* method of the *TouchEvent* class, which returns a *TouchPoint*, and then call the *getSceneX* and *getSceneY* methods of the *TouchPoint* class. These methods are also shown in Table 17.42.

Useful Classes and Methods for Mouse and Touch Events

Package **javafx.scene.input**

Methods of the *MouseEvent* Class

Return value **Method name and argument list**

`double` `getSceneX( )`

returns the *x* coordinate of this *MouseEvent* in the *Scene* where it occurs

`double` `getSceneY( )`

returns the *y* coordinate of this *MouseEvent* in the *Scene* where it occurs

`double` `getX( )`

returns the x coordinate of the *MouseEvent* relative to the node on which the mouse listener is registered

`double` `getY( )`

returns the y coordinate of the *MouseEvent* relative to the node on which the mouse listener is registered

Method of the *TouchEvent* Class

Return value **Method name and argument list**

`TouchPoint` `getTouchPoint( )`

returns the touch point for this *TouchEvent*

Methods of the *TouchPoint* Class

Return value **Method name and argument list**

`double` `getSceneX( )`

returns the *x* coordinate of this *TouchPoint* in the *Scene* in which it occurs

`double` `getSceneY( )`

returns the *y* coordinate of this *TouchPoint* in the *Scene* in which it occurs

`double` `getX( )`

returns the x coordinate of the *TouchPoint* relative to the node on which the touch listener is registered

`double` `getY( )`

returns the y coordinate of the *TouchPoint* relative to the node on which the touch listener is registered

Description

Our application will implement the *EventHandler* interface. The *EventHandler* interface uses generics. We specify the event type that it uses depending on the type of event that we want to handle. It includes only one method, *handle*, described in **Table 17.43**. Thus, the *EventHandler* interface is a functional interface, and we can use lambda expressions when implementing it.

The *handle* Method of the *EventHandler<T extends Event>* Interface

`void handle( T event )`

This method is called when an event occurs for which this handler is registered.

Description

The *Node* class includes many methods to set up event handling for mouse and touch events, as well as key events. Some of these methods are shown in **Table 17.44**. All the layout container classes, like *VBox* for example, inherit these methods from *Node*. The mouse-related methods accept a parameter of type *EventHandler <? super MouseEvent>*. *EventHandler* is an interface that uses generics. The syntax *<? super MouseEvent>* means that the type of event should be a *MouseEvent* or a superclass of the *MouseEvent* class. Similarly, the event handler for touch events accepts an event that is a *TouchEvent* or a superclass of *TouchEvent*.

Useful Methods of the *Node* Class for Mouse and Touch Events

Methods of the *Node* Class

Return value	Method name and argument

	list
`void`	`setOnMousePressed( EventHandler<? super MouseEvent> handler )`
registers *handler* as the listener to execute when the user presses the mouse. A parameter of *null* removes any previously registered handler.	
`void`	`setOnMouseMoved( EventHandler<? super MouseEvent> handler )`
registers *handler* as the listener to execute when the user moves the mouse. A parameter of *null* removes any previously registered handler.	
`void`	`setOnTouchPressed( EventHandler<? super TouchEvent> handler )`
registers *handler* as the listener to execute when the user touches the screen. A parameter of *null* removes any previously registered handler.	
`void`	`setOnTouchMoved( EventHandler<? super TouchEvent> handler )`
registers *handler* as the listener to execute when the user moves the touch point on the screen. A parameter of *null* removes any previously registered handler.	

Description

To illustrate mouse and touch events, we will build a simple submarine hunt game. A submarine is hidden somewhere in the window, and the user will try to sink the submarine by clicking the mouse or touching the screen at various locations in the window, simulating the dropping of a depth charge. Each time the user clicks the mouse or touches the screen, we will indicate how close that location is to the submarine. If the user's click or touch is too far from the submarine, we will display “In the water” in the title bar and draw a blue circle at the corresponding location. If the user's click or touch is close to the submarine, we will display “Close …” in the title bar. Finally, if the submarine is hit, we will change the title bar to “Sunk!”, display the submarine, and remove the listeners so the game ends.

Figure 17.22 shows the UML diagram for this application. We first code our Model, the *SubHunt* class (**Example 17.43**). It encapsulates the submarine hunt game and enables us to create a game of a given size, enable play, and enforce the rules of the game.

Figure 17.22 UML Diagram for the Sub Hunt Game

Description

Example 17.43 The *SubHunt* Class

```java
/* SubHunt class
*  Anderson, Franceschi
*/

import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import java.util.Random;

public class SubHunt
{
  public static int DEFAULT_GAME_SIZE = 300;
  public static int SIDE = 36; // size of submarine

  private int gameSize;
  private int xCtr; // x coordinate of center of submarine
  private int yCtr; // y coordinate of center of submarine
  private String status = "";
  private boolean hit;

  /** Constructor
  * @param gameSize the game size
  */
  public SubHunt( int gameSize )
  {
    if ( gameSize > SIDE )
       this.gameSize = gameSize;
    else
       this.gameSize = DEFAULT_GAME_SIZE;
    // generate submarine center
    Random random = new Random( );
    xCtr = SIDE / 2 + random.nextInt( this.gameSize - SIDE );
    yCtr = SIDE / 2 + random.nextInt( this.gameSize - SIDE );
    hit = false;
  }

  /** getStatus Accessor
  * @return status
  */
  public String getStatus( )
  {
    return status;
  }

  /** getGameSize Accessor
  * @return gameSize
  */
  public int getGameSize( )
  {
    return gameSize;
  }

  /** isHit method
  * @return hit, which indicates whether the sub has been hit
  */
  public boolean isHit( )
  {
    return hit;
  }

  /** play method
  * @param x the x coordinate of the play
  * @param y the y coordinate of the play
  */
  public void play( int x, int y )
  {
    // is click within the submarine?
    if ( Math.abs( x - xCtr ) < SIDE / 2
         && Math.abs( y - yCtr ) < SIDE / 2 )
    {
      status = "Sunk!";
      hit = true;
    }
    // is click close?
    else if ( Math.abs( x - xCtr ) < 2 * SIDE
              && Math.abs( y - yCtr ) < 2 * SIDE )
      status = "Close ...";
    // click is too far from submarine
    else
      status = "In the water";
  }

  /** draw method
  * @param gc a GraphicsContext object
  * @param x the x coordinate of the play
  * @param y the y coordinate of the play
  */
  public void draw( GraphicsContext gc, int x, int y )
  {
    if ( status.equals( "Sunk!" ) )
    {
      // draw sunken submarine
      gc.setFill( Color.BLACK );
      gc.fillRoundRect( xCtr - SIDE/2, yCtr - SIDE/2,
                        SIDE/2, SIDE, SIDE/2, SIDE/2 );

      gc.fillRoundRect( xCtr - SIDE/4, yCtr - SIDE/3,
                        SIDE/2, SIDE/2, SIDE/4, SIDE/4 );

      gc.strokeLine( xCtr + SIDE/4, yCtr - SIDE/9,
                     xCtr + SIDE/2, yCtr - SIDE/9 );

      // draw red depth charge
      gc.setFill( Color.RED );
      gc.fillOval( x - SIDE/2, y - SIDE/2, SIDE, SIDE );
    }
    else if ( status.equals( "In the water" ) ) // draw blue circle
    {
      gc.setFill( Color.BLUE );
      gc.fillOval( x - SIDE/2, y - SIDE/2, SIDE, SIDE );
    }
    // else Close ... , do not draw
  }
}
```

Description

Example 17.44 shows the *SubHuntViewController* class, which extends *VBox* (line 11). It combines the View and the Controller. At lines 13–15, we declare our instance variables. The *SubHunt* instance variable *sub* (line 13) represents the Model, the submarine hunt game that we display inside the window. At line 14, we include the instance variable *stage*, a reference to the *Stage*. We use *stage* to call the *setTitle* method (line 64) to update the title of the window each time the user plays. In the next example, we will show how to retrieve a reference to the *Stage* from the Controller class.

Example 17.44 The *SubHuntViewController* Class

```java
/* SubHuntViewController class
*  Anderson, Franceschi
*/

import javafx.scene.layout.*;
import javafx.scene.input.*;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.stage.Stage;

public class SubHuntViewController extends VBox
{
  private SubHunt sub; // submarine
  private Stage stage;
  private GraphicsContext gc;

  /** Constructor
  * @param sub the Model reference for this game
  * @param stage the Stage for this game
  * @param width the width of the Canvas rendering the game
  * @param height the height of the Canvas rendering the game
  */
  public SubHuntViewController( SubHunt sub,
                                Stage stage, int width, int height )
  {
    this.sub = sub;
    this.stage = stage;
    Canvas canvas = new Canvas( width, height );
    getChildren( ).add( canvas );
    gc = canvas.getGraphicsContext2D( );

    this.setOnMouseClicked( event -> handleMouseEvent( event ) );
    this.setOnTouchPressed( event -> handleTouchEvent( event ) );
  }

  /** handleMouseEvent method
  * @param event the MouseEvent generated by the user
  */
  public void handleMouseEvent( MouseEvent event )
  {
    int x = ( int ) event.getSceneX( );
    int y = ( int ) event.getSceneY( );
    play( x, y );
  }

  /** handleTouchEvent method
  * @param event the TouchEvent generated by the user
  */
  public void handleTouchEvent( TouchEvent event )
  {
    int x = ( int ) event.getTouchPoint( ).getSceneX( );
    int y = ( int ) event.getTouchPoint( ).getSceneY( );
    play( x, y );
  }

  /** play method
  * @param x the x coordinate of the play
  * @param y the y coordinate of the play
  */
  public void play( int x, int y )
  {
    sub.play( x, y );
    sub.draw( gc, x, y );
    stage.setTitle( sub.getStatus( ) );
    if ( sub.isHit( ) )
    {
      this.setOnMouseClicked( null );
      this.setOnTouchPressed( null );
    }
  }
}
```

Description

The constructor receives *SubHunt* and *Stage* references as parameters, which we store in the *sub* and *stage* instance variables at lines 26–27. When the user plays, we use the *sub* reference to call the various methods of the *SubHunt* class to enable play and enforce the rules of the game. When the user interacts with the View by clicking on the window, the Controller captures that information, calls the appropriate method of the *SubHunt* class, and updates the View accordingly. The constructor (lines 17–34) defines and displays the original View. It sets up event handling at lines 32–33 for mouse clicks and touch presses. At line 32, we use a lambda expression to define the listener for mouse events, which will execute the *handleMouseEvent* method.

The *handleMouseEvent* method (lines 36–44) retrieves the *x* and *y* coordinates of the mouse event (lines 41–42) and calls the *play* method (line 43) to enable play. On line 33, we again use a lambda expression to define the listener for touch events, which will execute the *handleTouchEvent* method (lines 46–54), which performs equivalent operations for a touch event.

Inside the *play* method (lines 56–70), we call the *play* method of the *SubHunt* model (line 62) to process the user's action, passing its location. Next, we update the View by drawing the current play (line 63). Then we update the title of the window by getting the updated status, and we display that status in the window title bar by calling the *setTitle* method (line 64). Finally, if the game is over (line 65), we disable event handling (lines 67–68) by passing a *null* argument to the *setOnMouseClicked* and *setOnTouchPressed* methods.

The *PlaySubHunt* class, shown in **Example 17.45**, instantiates a *SubHunt* object (line 16). It then instantiates a *SubHuntViewController* passing the *SubHunt* object we just instantiated as an argument, along with the stage and the game size. The controller extends *VBox*, so we set the controller as the *root* node for our *Scene*. **Figure 17.23** shows a run of this game. At this point, the user has sunk the submarine.

Example 17.45 The *PlaySubHunt* Class

```
/* PlaySubHunt class
   Anderson, Franceschi
*/

import javafx.application. Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class PlaySubHunt extends Application
{
  private final int GAME_SIZE = 500;

  @Override
  public void start( Stage stage )
  {
    SubHunt sub = new SubHunt( GAME_SIZE );
    SubHuntViewController root =
      new SubHuntViewController( sub, stage, GAME_SIZE, GAME_SIZE );

    Scene scene = new Scene( root, 500, 500 );
    stage.setTitle( "Play !!" );
    stage.setScene( scene );
    stage.show( );
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description

Figure 17.23 A Run of the *SubHunt* Game

Description

In the sub hunt game, we handled mouse clicks and touches. But we didn’t handle mouse movements or touch movements. To illustrate how to handle mouse and touch movements, we build a simple treasure hunt game that is similar to the submarine hunt game. A treasure is hidden somewhere in the window, and the user will try to find it by moving either the mouse or his or her finger inside the window. Depending on how close the mouse or touch is to the treasure, we display a message at the mouse or touch location so the user can eventually find the treasure and win the game. **Figure 17.24** shows the UML diagram for the treasure hunt game.

Figure 17.24 UML Diagram for the Treasure Hunt Game

Description

Again, we first code the Model, a class that encapsulates the functionality of our treasure hunt game (**Example 17.46**). It enables us to create a game of a given size, handle plays, determine when the game is over, and display the status of the game.

Example 17.46 The *TreasureHunt* Class

```
/* TreasureHunt class
*  Anderson, Franceschi
*/

import java.util.Random;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;

public class TreasureHunt
{
 public static int DEFAULT_GAME_SIZE = 300; // side of window
 public static int SIDE = 40; // side of treasure

 private int gameSize;
 private int xCtr; // x coordinate of center of treasure
 private int yCtr; // y coordinate of center of treasure
 private String status = ""; // message
 private boolean gameOver = false;

 /** Constructor
 * @param gameSize the size of the game
 */
 public TreasureHunt( int gameSize )
 {
  if ( gameSize > SIDE )
    this.gameSize = gameSize;
  else
    this.gameSize = DEFAULT_GAME_SIZE;
  // generate treasure center
  Random random = new Random( );
  xCtr = SIDE / 2 + random.nextInt( this.gameSize - SIDE );
  yCtr = SIDE / 2 + random.nextInt( this.gameSize - SIDE );
 }

 /** getGameSize accessor
 * @return gameSize
 */
 public int getGameSize( )
 {
  return gameSize;
 }

 /** isGameOver method
 * @return gameOver
 */
 public boolean isGameOver( )
 {
  return gameOver;
 }

 /** play method
 * @param x the x coordinate of the play
 * @param y the y coordinate of the play
 */
 public void play( int x, int y )
 {
  // is mouse within treasure?
  if ( Math.abs( x - xCtr ) < SIDE / 2
       && Math.abs( y - yCtr ) < SIDE / 2 )
  {
    gameOver = true;
    status = "Found";
  }
  // is mouse within half-length of the treasure?
  else if ( Math.abs( x - xCtr ) < ( 1.5 * SIDE )
            && Math.abs( y - yCtr ) < ( 1.5 * SIDE ) )
    status = "Hot";
  // is mouse within 1 length of the treasure?
  else if ( Math.abs( x - xCtr ) < ( 2 * SIDE )
            && Math.abs( y - yCtr ) < ( 2 * SIDE ) )
    status = "Warm";
  // is mouse within 2 lengths of the treasure?
  else if ( Math.abs( x - xCtr ) < ( 3 * SIDE )
            && Math.abs( y - yCtr ) < ( 3 * SIDE ) )
    status = "Lukewarm";
  else // mouse is not near treasure
    status = "Cold";
 }

 /** draw method
 * @param gc a GraphicsContext reference
 * @param x the x coordinate of the play
 * @param y the y coordinate of the play
 */
 public void draw( GraphicsContext gc, int x, int y )
 {
  gc.setFill( Color.BLUE );
  gc.setFont( new Font( 24 ) );
  gc.clearRect( 0, 0, gameSize, gameSize );
  if ( status.equals( "Found" ) ) // if found, draw treasure
  {
   gc.setFont( new Font( 16 ) );
   gc.setFill( Color.RED );
   gc.fillRect( xCtr - SIDE / 2, yCtr - SIDE / 2, SIDE, SIDE );
   gc.setFill( Color.GREEN );
   gc.fillText( "$$$", xCtr - SIDE / 3, yCtr + SIDE / 6 );
  }
  else
   gc.fillText( status, x, y ); // display current status
 }
}
```

Description

The *status* (line 18) instance variable, a *String*, stores the state of the player's position compared to the location of the treasure. We also define a *boolean* flag variable, *gameOver*, which is initially *false* (line 19); when the user finds the treasure, we will change its value to *true*.

In the constructor, we randomly generate the (*x*, *y*) coordinate for the center of the treasure and store the generated values in the *xCtr* and *yCtr* instance variables (lines 30–33). The *play* method (lines 52–79) sets the value of *status*. If the player is far from the treasure, we set the value of *status* to "Cold." As the user moves closer and closer to the treasure, the value becomes "Lukewarm," then "Warm," and then "Hot," and finally "Found" when the player finds the treasure. If the treasure is found, we set the value of *gameOver* to *true* (line 62).

Like the sub hunt game, the *draw* method (lines 81–101) draws what is happening in the game. We first test if the value of *status* is "Found" (line 91). If it is, the treasure has been found and we reveal its location by drawing it (lines 93–97). If the treasure has not been found, we display *status* at the current mouse or touch location (line 100), determined by the parameters *x* and *y* sent to the *draw* method.

Example 17.47 shows the *TreasureHuntViewController* class. Like the sub hunt example, it combines the View and the Controller. The constructor (lines 19–35) defines and displays the View in a *Canvas* (lines 29–30). The Model is represented by the instance variable *hunt* (line 16), which stores the *TreasureHunt* object sent to the constructor from the *start* method (line 26). In this application, instead of coding the event handler as a *private* inner class, we define our application class as implementing the *EventHandler* interface. As a result, our application is a listener, and we register the listener on itself. Thus, in our class definition, we include the clause *implements EventHandler<InputEvent>* (line 13). We choose the *InputEvent* class because it is a superclass of both *MouseEvent* and *TouchEvent*, and thus we can register both mouse and touch events on our *TreasureHuntViewController*. At lines 33–34, we register this *TreasureHuntViewController* object on itself as an *EventHandler* for both mouse and touch movements.

Example 17.47 The *TreasureHuntViewController* Class

```java
/* TreasureHuntViewController class
*  Anderson, Franceschi
*/

import javafx.event.*;
import javafx.scene.layout.*;
import javafx.scene.input.*;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.stage.Stage;

public class TreasureHuntViewController extends VBox
       implements EventHandler<InputEvent>
{
  private int width, height;
  private TreasureHunt hunt;
  private GraphicsContext gc;

  /** Constructor
  * @param hunt the model reference
  * @param width the width of the Canvas rendering the game
  * @param height the height of the Canvas rendering the game
  */
  public TreasureHuntViewController( TreasureHunt hunt, int width, int height )
  {
   this.hunt = hunt;
   this.width = width;
   this.height = height;
   Canvas canvas = new Canvas( width, height );
   getChildren( ).add( canvas );
   gc = canvas.getGraphicsContext2D( );

   this.setOnMouseMoved( this );
   this.setOnTouchMoved( this );
  }

  /** handle method
  * @param event the InputEvent generated by the user
  */
  public void handle( InputEvent event )
  {
   int x = -100, y = -100;
   if ( event.getEventType( ) == MouseEvent.MOUSE_MOVED )
   {
     MouseEvent mEvent = ( MouseEvent ) event ;
     x = ( int ) mEvent.getSceneX( );
     y = ( int ) mEvent.getSceneY( );
   }
   else if ( event.getEventType( ) == TouchEvent.TOUCH_MOVED )
   {
    TouchEvent tEvent = ( TouchEvent ) event ;
    x = ( int ) tEvent.getTouchPoint( ).getSceneX( );
    y = ( int ) tEvent.getTouchPoint( ).getSceneY( );
   }

   hunt.play( x, y );

   hunt.draw( gc, x , y );
   if ( hunt.isGameOver( ) )
   {
     this.setOnMouseMoved( null );
     this.setOnTouchMoved( null );
     Stage stage = ( Stage ) getScene( ).getWindow( );
     stage.setTitle( "Thank you for playing" );
   }
  }
}
```

Description

In this game, the events we want to handle are the user moving the mouse or his or her finger on the screen. Because *TreasureHuntViewController* implements the *EventHandler* interface, we must provide a *handle* method, which we do at lines 37–66. Because the ways we retrieve the *x* and *y* coordinates of a mouse event and a touch event are different, we first test if the event is a mouse event (line 43) or a touch event (line 49) by calling the *getEventType* method, inherited from the *Event* class. We compare the return value to the appropriate constant of either the *MouseEvent* class or the *TouchEvent* class. Both constants are shown in **Table 17.45**. We type cast the event to the appropriate event at lines 45 and 51, then retrieve the *x* and *y* coordinates of the mouse or touch movement.

Useful *EventType* Constants of the *MouseEvent* and *TouchEvent* Classes

Class	Constant
`MouseEvent`	`MOUSE_MOVED`
constant identifying a mouse move event	
`TouchEvent`	`TOUCH_MOVED`
constant identifying a touch move event	

Description

Then we call the *play* method of the *TreasureHunt* model (line 56) to process the play. Next, we display the result of the current move (line 58) by calling the *draw* method of *TreasureHunt*, passing the *GraphicsContext* reference *gc* and the mouse or touch location. If the game is over (i.e., the treasure has been found), we remove this object, the current *TreasureHuntViewController*, as a listener (lines 61–62) for both mouse and touch moving events.

At lines 63–64, we update the title of the window to reflect the fact that the user has found the treasure and the game is over. In order to do that, we need a reference to the *Stage* of this *VBox*, so we can call the *setTitle* method. The *getScene* method, inherited by *VBox* from the *Node* class, returns the *Scene* containing this *Node*, the *TreasureHuntViewController*. With it, we call the *getWindow* method in order to get a reference to the window, which represents the stage. *Stage* is a subclass of *Window*, so we cast the return value of *getWindow* to a *Stage* in order to get a reference to the stage. **Table 17.46** shows the *getScene* and *getWindow* methods.

Useful Classes and Methods to Retrieve the Stage

Class	Method
`Node`	`Scene getScene( )`
returns the *Scene* containing this *Node*	
`Scene`	`Window getWindow( )`
returns the window for this *Scene*	

Description

Finally, the *PlayTreasureHunt* class, shown in **Example 17.48**, includes the *main* method to create a *TreasureHuntViewController* application showing a *TreasureHunt* game. **Figures 17.25** and **17.26** show the program running. In Figure 17.25, the user is getting close to the treasure, and in Figure 17.26, the user has found the treasure.

Example 17.48 The *PlayTreasureHunt* Class

```
/* PlayTreasureHunt class
   Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class PlayTreasureHunt extends Application
{
  private final int GAME_SIZE = 500;

  @Override
  public void start( Stage stage )
  {
    TreasureHunt th = new TreasureHunt( GAME_SIZE );
    TreasureHuntViewController root =
      new TreasureHuntViewController( th, GAME_SIZE, GAME_SIZE );

    Scene scene = new Scene( root, GAME_SIZE, GAME_SIZE );
    stage.setTitle( "Play !!" );
    stage.setScene( scene );
    stage.show( );
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description

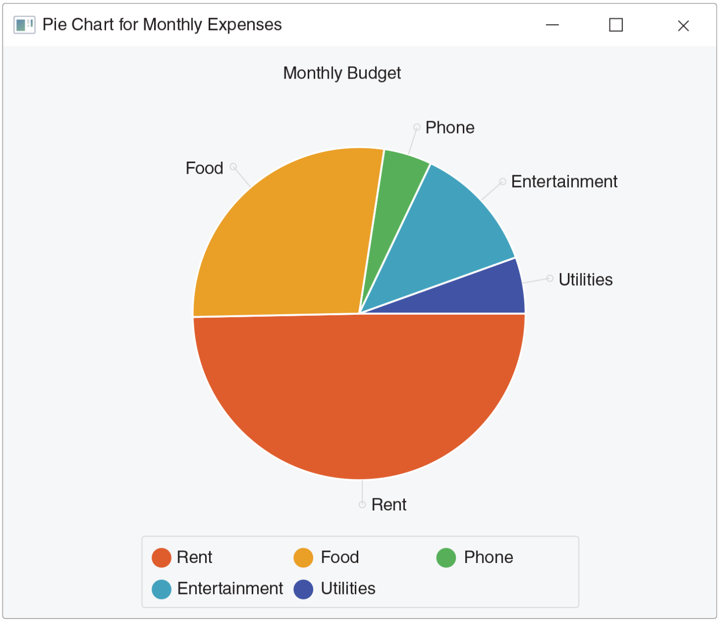
Figure 17.25 The User Is Getting Close to the Treasure

Description

Figure 17.26 The User Has Found the Treasure

Description

In both the sub hunt and treasure hunt examples, the Model is reusable in another application. However, the View and the Controller are combined in the same class and are not reusable. Later in the chapter, we will make the View reusable too.


Pie Chart for Monthly Expenses
Monthly Budget
Phone
Food
Entertainment
Utilities
Rent
● Rent
● Food
● Phone
● Entertainment
● Utilities

17.15 Using a List to Display a Pie Chart

JavaFX includes classes that make it easy to display various types of charts, including line charts, pie charts, and bar charts. To display a chart, we first create a list that stores the data we want to display. We then assign that data to the appropriate chart object. After we include the chart object in a scene, the chart draws itself automatically. The *ObservableList* interface, from the *javafx.collections* package, is a list that can be linked to a GUI component, such as a *ComboBox*. Whenever the values in the list change, the associated GUI component is automatically updated to reflect those changes. Several chart classes, such as *PieChart* and *BarChart*, can be constructed by linking their data to an *ObservableList*.

In this example, we use the *PieChart* class to display a pie chart, a filled circle composed of colored slices, where each slice represents a percentage of the whole pie. The *PieChart* constructor, listed in **Table 17.47**, accepts an *ObservableList* of *PieChart.Data* as its only parameter. *PieChart.Data* is an inner class of *PieChart* and each object represents one slice of the pie chart. A slice is defined by its label and its value. The *PieChart.Data* constructor shown in Table 17.47 accepts a *String* parameter for the label and a *double* parameter for the value.

PieChart.Data and *PieChart* Constructors

Package	javafx.scene.chart
***PieChart.Data* and *PieChart* Constructors**	
`PieChart.Data( String name, double value )`	
creates a *PieChart.Data*, a slice of a *PieChart*; the slice value is *value* and its label is *name*	
`PieChart( ObservableList<PieChart.Data> data )`	
creates a *PieChart* using *data* as its data for all its slices	

Description

To display a pie chart, we do the following:

- Create a list of *PieChart.Data* objects.
- Convert that list to an *ObservableList*.
- Create a *PieChart* associated with that *ObservableList*.
- Create a *Scene* with that *PieChart*.

Example 17.49 creates a pie chart displaying the various parts of a monthly budget so we can visualize the percentage of the total for each category. At lines 11–14, we declare two arrays, *categories* and *expenses*, which hold the labels and the corresponding values for the pie chart slices. At lines 19–22, we create and fill *list*, an *ArrayList* of *PieChart.Data*. Each *PieChart.Data* is defined with a label from the *categories* array and a value from the *expenses* array (line 22). At lines 23–24, we call the *observableList* method of the *FXCollections* class, shown in **Table 17.48**, to construct the *ObservableList* *pieChartData* from that *ArrayList*. The *ArrayList* class *implements* the *List* interface, and thus an *ArrayList* can be used as an argument of the *observableList* method.

Example 17.49 An Application to Display a Pie Chart

```java
import java.util.ArrayList;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.chart.*;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class MonthlyBudgetPieChart extends Application
{
  private String [ ] categories
    = { "Rent", "Food", "Phone", "Entertainment", "Utilities" };
  private double [ ] expenses
    = { 800, 450, 72.50, 200, 90 };

  @Override
  public void start( Stage stage )
  {
    // Create a PieChart.Data of monthly expenses
    ArrayList<PieChart.Data> list = new ArrayList<PieChart.Data>( );
    for ( int i = 0; i < expenses.length; i++ )
      list.add( new PieChart.Data( categories[i], expenses[i] ) );
    ObservableList<PieChart.Data> pieChartData
      = FXCollections.observableList( list );

    // Create a PieChart with pieChartData
    PieChart chart = new PieChart( pieChartData );
    chart.setTitle( "Monthly Budget" );

    // Define the scene and stage
    Scene scene = new Scene( chart, 500, 400 );
    stage.setScene( scene );
    stage.setTitle( "Pie Chart for Monthly Expenses" );
    stage.show( );
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description

The *observableList* Method of the *FXCollections*

The *observableList static* Method of the *FXCollections* Class

Return value	Method name and argument list
`ObservableList<E>`	`observableList( List<E> list )`
constructs and returns an *ObservableList* composed of elements of type E from *list*. Changes to that *ObservableList* will be reflected in components that are registered on that *ObservableList*.	

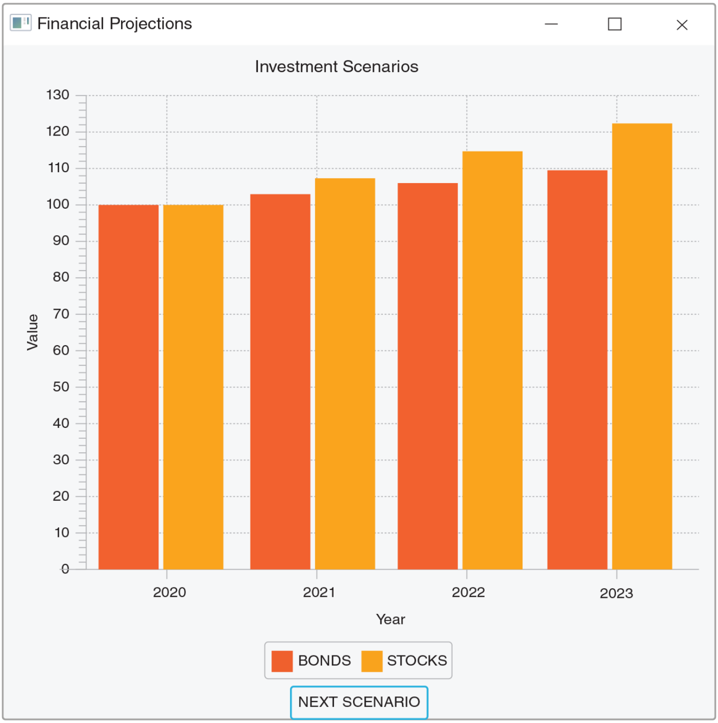
Description

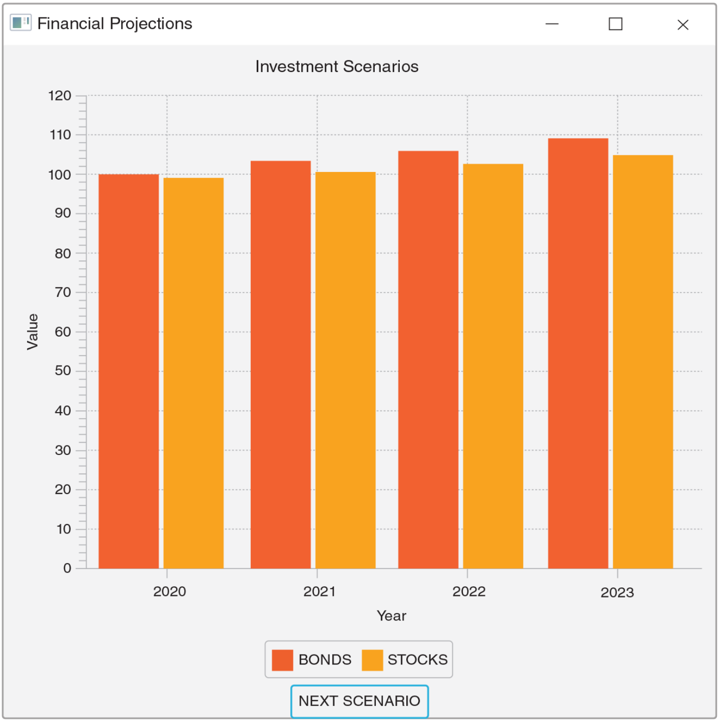
At lines 26–28, we construct the *PieChart* chart with *pieChartData* and set its title. At lines 30–34, we construct a *Scene* with the *PieChart* (line 31) and set the *Stage* with that *Scene*. The *PieChart* class inherits from the *Parent* class, and thus a *PieChart* can be used to construct a scene. **Figure 17.27** shows the pie chart.

Figure 17.27 Displaying a Pie Chart for a Monthly Budget

Description

In the next example, we show how a graphical component redraws itself automatically when the observable list that it is bound to changes.


Financial Projections
Investment Scenarios
Value
130
120
110
100
90
80
70
60
50
40
30
20
10
0
2020
2021
2022
2023
Year
BONDS
STOCKS
NEXT SCENARIO


Financial Projections
Investment Scenarios
120
110
100
90
80
70
60
50
40
30
20
10
0
Value
2020
2021
2022
2023
Year
BONDS
STOCKS
NEXT SCENARIO

17.16 Using a List to Display a Dynamic Bar Chart

In this example, we display a bar chart that displays projected values of stocks and bonds over 4 years for an initial investment of $100. The *BarChart* class can be used to create a two-dimensional bar chart. A *BarChart* is defined using a horizontal axis, a vertical axis, and a list consisting of one or two *series* of data, each series representing a single-dimensional list of data. Thus, if we use only one series in that list, we can define a one-dimensional bar chart as a special case of a two-dimensional bar chart. Our example uses two series of data, one for stocks and one for bonds. The projected values of the bonds are fixed, but the projected values of the stocks toggle between two sets of values, depending on what growth rate we use. The user can toggle between the two sets of values by clicking on a button. The *BarChart* is bound to an *ObservableList*. When the user clicks on the button, we change the data inside the

ObservableList to the alternate set of values. Because the *BarChart* is bound to the *ObservableList*, it is automatically updated. **Figures 17.28** and **17.29** show both states of the *BarChart*.

Figure 17.28 The Stocks and Bonds Using the First Set of Values for Stocks

Description

Figure 17.29 The Stocks and Bonds Using the Second Set of Values for Stocks

Description

Before explaining how we can create and draw a bar chart, we define our Model. It includes three classes:

- *Investment*, an *abstract* superclass: An *Investment* has a type (Stocks or Bonds), an original amount, a growth rate, and the number of years over which to compute the value of the original amount as we apply the growth rate.
- *Bonds*, a subclass of *Investment*: The growth rate is constant.
- *Stocks*, a subclass of *Investment*: The growth rate toggles between two constant values.

Examples 17.50, **17.51**, and **17.52** show these three classes. The *updateAmounts* method (lines 76–83) of the *Investment* class updates the instance variable *amounts* based on the growth rate. *Investment* specifies the *update* method as *abstract*. It is meant to update the growth rate and then update the resulting cash flow. In this example, a *Bonds* has a constant growth rate; thus, its *update* method (lines 19–25 of Example 17.51) does nothing, that is, it leaves the growth rate unchanged. The *update* method of the *Stocks* class (lines 20–31 of Example 17.52) makes the growth rate of a *Stocks* investment toggle between two constant values (lines 7–8).

Example 17.50 The *Investment* Class

```
/** abstract Investment class
*   Anderson, Franceschi
*/

public abstract class Investment
{
  public static final int DEFAULT_YEARS = 4;
  public static final double BEGIN_INVEST_AMOUNT = 100;
  private double [ ] amounts;
  private int years;
  private double growthRate;
  private String type;

  /* Constructor
  * @param years, an int, the new value of years
  * @param type, a String, the new value for the investment type
  */
  public Investment( int years, String type )
  {
    if ( years < 1 )
      this.years = DEFAULT_YEARS;
    else
      this.years = years;

    amounts = new double[this.years];
    amounts[0] = BEGIN_INVEST_AMOUNT;

    this.type = type;
  }

  /* getAmounts method
  * @return a copy of amounts, a double [ ]
  */
  public double [ ] getAmounts( )
  {
    double [ ] temp = new double[amounts.length];
    for ( int i = 0; i < amounts.length; i++ )
        temp[i] = amounts[i];
    return temp;
  }

  /* Accessor method for years
  * @return number of years
  */
  public int getYears( )
  {
    return years;
  }

  /* Accessor method for type
  * @return investment type
  */
  public String getType( )
  {
    return type;
  }

  /* Accessor method for growthRate
  * @return growthRate
  */
  public double getGrowthRate( )
  {
    return growthRate;
  }

  /* Mutator method for growthRate
  * @param growthRate, the new value for growthRate
  * @return a reference to this object
  */
  public Investment setGrowthRate( double growthRate )
  {
    this.growthRate = growthRate;
    return this;
  }

  /* updateAmounts method
  * updates amounts based on growthRate
  */
  public void updateAmounts( )
  {
    for ( int i = 1; i < amounts.length; i++ )
      amounts[i] = amounts[i-1] * ( 1 + growthRate );
  }

  /* update abstract method
  */
  public abstract void update( );
}
```

Description

Example 17.51 The *Bonds* Class

```
/** Bonds class
*   Anderson, Franceschi
*/

public class Bonds extends Investment
{
  private final double BOND_GROWTH_RATE = .03;

  /* Constructor
  * @param years the new number of years
  */
  public Bonds( int years )
  {
    super( years, "Bonds" );
    setGrowthRate( BOND_GROWTH_RATE );
    updateAmounts( );
  }

  /* update method
  * do nothing method: Leaves the growth rate unchanged
  */
  public void update( )
  {
    // growth rate does not change
  }
}
```

Description

Example 17.52 The *Stocks* Class

```java
/** Stocks class
*   Anderson, Franceschi
*/

public class Stocks extends Investment
{
  private static final double SLOW_GROWTH_RATE = .02;
  private static final double FAST_GROWTH_RATE = .07;

  /* Constructor
  * @param years the new number of years
  */
  public Stocks( int years )
  {
    super( years, "Stocks" );
    setGrowthRate( SLOW_GROWTH_RATE );
    updateAmounts( );
  }

  /* update method
  *  toggles the growth rate and updates amounts accordingly
  */
  public void update( )
  {
    if ( Math.abs( getGrowthRate( ) - SLOW_GROWTH_RATE ) < 0.0001 )
      setGrowthRate( FAST_GROWTH_RATE );
    else
      setGrowthRate( SLOW_GROWTH_RATE );

    updateAmounts( );
  }
}
```

Description

The *XYChart* class, which uses generics, is the base class for charts involving two axes. *XYChart* is the direct superclass of *BarChart*. The *XYChart* class provides the functionality to draw the two axes and the contents of the chart. It contains two inner classes:

- *XYChart.Data*, which encapsulates a single data item to be used in two-axis charts
- *XYChart.Series*, a series, or list, of *XYChart.Data* items

XYChart, *BarChart*, *XYChart.Data*, and *XYChart.Series* all use two types of objects; the first object serves as a label for the second. Thus, often, the first type is *String* and the second type is *Number*. The *Number* class is the superclass for all numeric wrapper classes (*Integer*, *Double*, ..) and thus can be used with all primitive integer data types since Java supports both autoboxing (automatic conversion from a primitive type to a wrapper type) and unboxing (automatic conversion from a wrapper type to a primitive type).

Typically, the observable list bound to a bar chart is a list of lists, that is, it is a two-dimensional data structure. In our example, the observable list contains a list of *XYChart.Series* items. Each *XYChart.Series* is a list of *XYChart.Data* items. Each *XYChart.Data* item contains two elements: the year, which is a *String*; and the value, which is a *double*. The *String* serves as a label for the value. For example, our initial two *XYChart.Series* contain the following data:

Bonds series: “2020”,100.0 “2021”,103.0 “2022”,106.09 “2023”,109.2727
Stocks series: “2020”,100.0 “2021”,107.0 “2022”,114.49 “2023”,122.5043

Description

The first series is the bonds series, which contains four data items; each data item includes a year and a value. The second series is the stocks series and also contains four data items. Both series are parallel to each other; not only do the two series contain the same number of items, four, but the value of the year for two corresponding data items is the same. In this way, the bar chart not only displays the values, but also enables a quick visual comparison between the two series.

Example 17.53 shows our *BarChartView* class. It extends *BorderPane* and contains a *BarChart* in its center and a button at the bottom. The *BarChart bc* and the *ArrayList series* are defined at lines 20 and 22. The *addBarChart* method (lines 40–71), called by the constructor at line 37, creates the bar chart and places it in the center position of the *BorderPane*. Inside the *addBarChart* method, we do the following:

Example 17.53 A View Class Displaying a Bar Chart

```
/** BarChartView class
* Layout using a BorderPane's
* bottom and center positions.
* The bottom HBox is made up of one button.
* The center position is made up of a BarChart.
* Anderson, Franceschi
*/

import java.util.ArrayList;
import javafx.event.*;
import javafx.geometry.Pos;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.chart.*;

public class BarChartView extends BorderPane
{
  private HBox buttonPanel; // bottom, holds button
  private Button button;
  private BarChart<String, Number> bc; // center, holds chart

  ArrayList<XYChart.Series<String, Number>> series;
  private int startingYear;

  public BarChartView( int startingYear, Investment [ ] investments )
  {
    super( );
    this.startingYear = startingYear;

    buttonPanel = new HBox( );
    button = new Button( );
    button.setText( "NEXT SCENARIO" );
    buttonPanel.setAlignment( Pos.CENTER );
    buttonPanel.getChildren( ).add( button );
    setBottom( buttonPanel );

    addBarChart( investments );
  }

  public void addBarChart( Investment [ ] investments )
  {
    // Create a BarChart for investment
    CategoryAxis xAxis = new CategoryAxis( );
    NumberAxis yAxis = new NumberAxis( );
    bc = new BarChart<String, Number>( xAxis, yAxis );
    bc.setTitle( "Investment Scenarios" );
    xAxis.setLabel( "Year" );
    yAxis.setLabel( "Value" );

    // Create an ArrayList of XYChart.Series using investments data
    series = new ArrayList<XYChart.Series<String, Number>>( );

    for ( int i = 0; i < investments.length; i++ )
    {
      XYChart.Series<String, Number> currentInvestmentSeries
        = new XYChart.Series<String, Number>( );
      currentInvestmentSeries.setName( investments[i].getType( ) );

      double [ ] currentAmounts = investments[i].getAmounts( );
      for ( int j = 0; j < currentAmounts.length; j++ )
      {
        currentInvestmentSeries.getData( ).add(
           new XYChart.Data<String, Number>(
             String.valueOf( startingYear + j ), currentAmounts[j] ) );
      }
      series.add( currentInvestmentSeries );
    }
    // Fill BarChart with XYChart.Series array
    bc.getData( ).addAll( series );
    setCenter( bc );
  }

  public void update( Investment [ ] investments )
  {
    // update investments
    for ( int i = 0; i < investments.length; i++ )
    {
      // do not update bonds series
      if ( ! investments[i].getType( ).equals( "Bonds" ) )
      {
        // retrieve series
        XYChart.Series<String, Number> currentInvestmentSeries
          = series.get( i );

        // change data of series
        currentInvestmentSeries.getData( ).clear( );
        double [ ] currentAmounts = investments[i].getAmounts( );
        for ( int j = 0; j < currentAmounts.length; j++ )
        {
           currentInvestmentSeries.getData( ).add(
             new XYChart.Data<String, Number>( ""
               + ( startingYear + j ), currentAmounts[j] ) );
        }

        // update series
        series.set( i, currentInvestmentSeries );
      }
    }
  }

  public void setButtonListener( EventHandler<ActionEvent> eh )
  {
    button.setOnAction( eh );
  }
}
```

Description

1. Instantiate an empty *BarChart* with an *x* axis and a *y* axis (lines 42–48).
2. Create an *ArrayList* of *XYChart.Series* objects (lines 50–67).
3. Access the data of the *BarChart* and add the *ArrayList* to it (lines 68–69).
4. Place the *BarChart* at the center of the *BorderPane* (line 70).

Table 17.49 lists one of the *BarChart* constructors. Since the constructor accepts two *Axis* parameters, we first need to create two axes for our bar chart. The *CategoryAxis* and *NumberAxis* classes are subclasses of the *abstract Axis class*:

- *CategoryAxis* encapsulates an axis that displays categories where each value represents a unique category (tick mark) along the axis. In this example, we use a *CategoryAxis* to show the years.
- *NumberAxis* encapsulates an axis that plots a range of numbers with major tick marks every “tickUnit.” We can use any *Number* type with this axis, *Integer, Double*, etc. … In this example, we use a *NumberAxis* for the values.

A Constructor of the *BarChart* Class

Package	**javafx.scene.chart**
`BarChart( Axis<X> xAxis, Axis<Y> yAxis )` creates a *BarChart* with the two axes. One of the axes should be a *CategoryAxis*, the other one a *ValueAxis*. Their order determines if the bar chart is horizontal or vertical.	

Description

At lines 42–44, we create the two axes and the *BarChart*. We set the title of the bar chart at line 46 and the labels of the axis at lines 47–48.

Next, we instantiate *series* (line 51), an *ArrayList* of *XYChart.Series<String, Number>*. For each *Investment* reference in *investments*, we create an *XYChart.Series* and add it to *series*. **Table 17.50** shows an *XYChart.Series* constructor, as well as an *XYChart.Data* constructor. There are only two elements in *investments* in this example, one *Bonds* object and one *Stocks* object. At lines 55–56, we instantiate an *XYChart.Series<String, Number>* named *currentInvestmentSeries*. At line 57, we set the name that will be displayed for this *currentInvestmentSeries* to the type of investment of the current *Investment* element. To do this, we call the *setName* method shown in **Table 17.51**. Next, we set the values in *currentInvestmentSeries* to the values in the current *Investment* element. After we retrieve the array of values for the current *Investment* element and assign it to the array *currentAmounts*, we loop through *currentAmounts* at lines 60–65 and create an *XYChart.Data* item for each value in *currentAmounts* (lines 63–64). We set the label of each *XYChart.Data* item to the current year and its value to the current amount. We compute the current year by adding the counting index of the loop, *j*, to *startingYear*. After the loop, we add *currentInvestmentSeries* to the *series ArrayList* at line 66. At line 69, we call *getData* to retrieve the *ObservableList* associated with *bc*. At this point, the observable list is empty, so we call the *addAll* method to set the observable list to *series*. When we add *bc* to the center of the *BorderPane* at line 70, the bar chart is displayed.

XYChart.Series and *XYChart.Data* Constructors

Package **javafx.scene.chart**

`XYChart.Series( )`

creates an *XYChart.Series* that will contain *XYChart.Data* items

`XYChart.Data( X xValue, Y yValue )`

creates data for the two axes of the chart; the X and Y types should match the X and Y types for the *x* and *y* axes

Description

The *getData* and *setName* Methods of the *XYChart.Series* Class

Return Value	Method name and argument list
`ObservableList<XYChart.Data<X, Y>>`	`getData( )`
returns the *ObservableList* of data items that make up this series	
`void`	`setName( String value )`
sets the name to be displayed for this series	

Description

The *update* method (lines 73–99) is called by the controller when the view needs to be updated because the user updated the data in the model. The *update* method updates the data inside the *ArrayList series* based on the state of the model. Since the *BarChart bc* is bound to *series*, the bar chart will be updated automatically. Since bond values do not change, we only update the data for investments that are not bonds (line 79) when we loop through *investments* at lines 76–98. We first retrieve the current *XYChart.Series* at lines 81–83 and assign it to the reference *currentInvestmentSeries*. Because we add new *XYChart.Data* items storing the updated data to *currentInvestmentSeries* (lines 90–92), we first need to empty *currentInvestmentSeries* by calling the *clear* method (line 86). If we do not, we would be adding new *XYChart.Data* items to the existing ones. At lines 95–96, we update the current element of *series*.

The *setButtonListener* method (lines 101–104), meant to be called by the controller, sets up event handling for the button.

Example 17.54 shows the *BarChartViewController* class, the controller for our bar chart application. It includes the instance variables *investments* and *view* (lines 10–11), which represent the View and the Model for the application, respectively. The *BarChartViewController* class implements *EventHandler* and is its own event handler. We set up event handling at line 18 and handle the event in the *handle* method (lines 21–28) by first updating the Model (lines 23–25) and then updating the View (lines 26–27).

Example 17.54 The Controller for the Bar Chart Application

```
 1 /** BarChartViewController class
 2 *   Anderson, Franceschi
 3 */
 4
 5 import javafx.event.*;
 6 import javafx.scene.control.*;
 7
 8 public class BarChartViewController implements EventHandler<ActionEvent>
 9 {
10   private BarChartView view;
11   private Investment [ ] investments;
12
13   public BarChartViewController( BarChartView view,
14                                  Investment [ ] investments )
15   {
16     this.investments = investments;
17     this.view = view;
18     ( this.view ).setButtonListener( this );
19   }
20
21   public void handle( ActionEvent event )
22   {
23     // update the model
24     for ( int i = 0; i < investments.length; i++ )
25       investments[i].update( );
26     // update the view
27     view.update( investments );
28   }
29 }
```

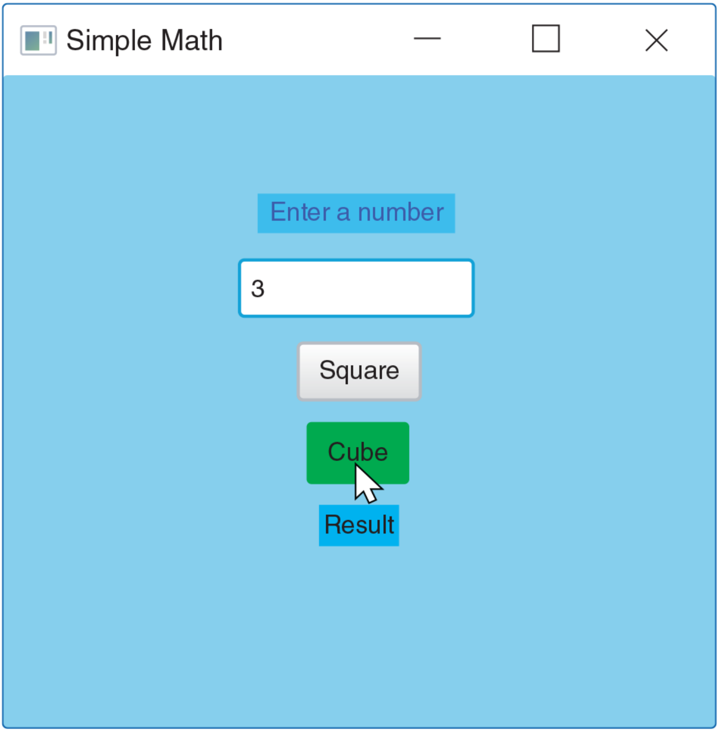
Description

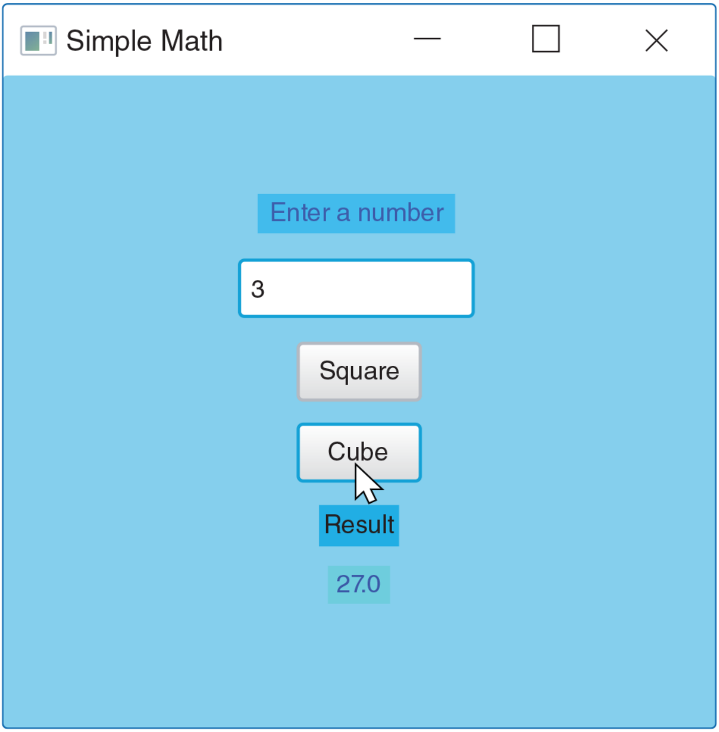
Example 17.55 shows the *InvestmentApplication* class. We create an array of two *Investment* objects at lines 14–18, create the View for the app at line 19, and create the Controller for the application at lines 27–28. We specify that *2020* is the starting year for our data (line 19).

Example 17.55 The *InvestmentApplication.java* File

```
/** InvestmentApplication class
*   Anderson, Franceschi
*/

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class InvestmentApplication extends Application
{
  @Override
  public void start( Stage stage )
  {
    int years = 4;
    Stocks stocks = new Stocks( years );
    Bonds bonds = new Bonds( years );
    Investment [ ] investments = { bonds, stocks };

    BarChartView root = new BarChartView( 2020, investments );

    Scene scene = new Scene( root, 540, 510 );
    stage.setTitle( "Financial Projections" );
    stage.setScene( scene );
    stage.show( );

    BarChartViewController controller
      = new BarChartViewController( root, investments );
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description


Simple Math
Enter a number
3
Square
Cube
Result

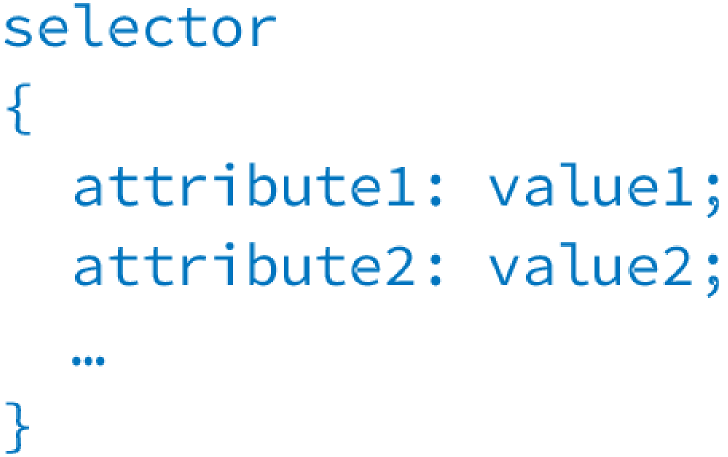


17.17 Using a Style Sheet to Style the View

GUI components come with a default style. For example, the background color is white and the text is black. Labels have a rectangular shape, and buttons have a rectangular shape with rounded corners. We can create our own style and customize the look and feel of our application. It is typical to define a style in a separate file, thus separating the contents of the GUI from its style. In this way, it is easier to edit and separately maintain the contents of the GUI and its style; we can modify the style file, also called a **style sheet**, and the new style is automatically applied to the app.

Earlier in the chapter we introduced CSS. JavaFX supports **Cascading Style Sheets** (**CSS**), a language that describes how a document should be styled. CSS is often used with web pages, and the JavaFX version of CSS is similar to the CSS used with web pages.

The general syntax to style a GUI component in a style sheet is:



```
selector
{
   attribute1: value1;
   attribute2: value2;

   …
}
```

Description

The *selector* can be a GUI component, such as a *Label*, but it can also be something identifying a group of components or a specific component. Attributes are prefixed with *-fx-*.

To illustrate this, we add styles to our Simple Math application, shown previously in Examples 17.5 through 17.8. **Example 17.56** shows a style sheet for the Simple Math application. At lines 6–10, we style all the labels so their background color is deepskyblue and their foreground color is blue. If we want to give a specific style to a single component, we can give it an id in the FXML file and style that id in the style sheet. In Example 17.6, we gave the id result to the bottom label, which displays the result of the Math operation. To style that label differently from the other labels, we define a style for its id. A selector for an id starts with the # sign followed by the id. At lines 12–16, we style the component with id result so that its background color is aqua (line 14) and its text is bold (line 15). Note that the background color of the bottom label is styled twice: once at line 8 where all the labels are styled, and another time at line 14 where the id result is styled. The id's style prevails over the component's general style. Thus, the background color of the bottom label is aqua. However, because the id's style does not redefine the foreground color, the foreground color defined for all the labels at line 9 applies to the bottom label.

Finally, at lines 18–21, we define the background color of the buttons when the mouse hovers over them as green.

Example 17.56 A JavaFX Style Sheet for the Simple Math Application

```
 1 VBox
 2 {
 3   -fx-background-color: skyblue;
 4 }
 5
 6 Label
 7 {
 8   -fx-background-color: deepskyblue;
 9   -fx-text-fill: blue;
10 }
11
12 #result
13 {
14   -fx-background-color: aqua;
15   -fx-font-weight: bold;
16 }
17
18 Button:hover
19 {
20   -fx-background-color: #00FF00;
21 }
```

Description

To apply our style sheet to our application, we link the style sheet to the scene. **Example 17.57** shows the modified *SimpleMathPractice* class. We link the style sheet at line 23 by calling the *getStylesheets* method of the *Scene* class. If we have more than one style sheet, we can add as many as desired using this method.

Example 17.57 Importing a Style Sheet

```
/*  Simple Math Operations Using Buttons
    Anderson, Franceschi
*/

import java.net.URL;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.layout.VBox;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class SimpleMathPractice extends Application
{
  @Override
  public void start( Stage stage ) // throws Exception
  {
    try
    {
      URL url =
         getClass( ).getResource( "fxml_simple_math.fxml" );
      VBox root = FXMLLoader.load( url );
      Scene scene = new Scene( root, 300, 275 );
      scene.getStylesheets( ).add( "simple_math.css" );
      stage.setTitle( "Simple Math" );
      stage.setScene( scene );
      stage.show( );
    }
    catch ( Exception e )
    {
      e.printStackTrace( );
    }
  }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description

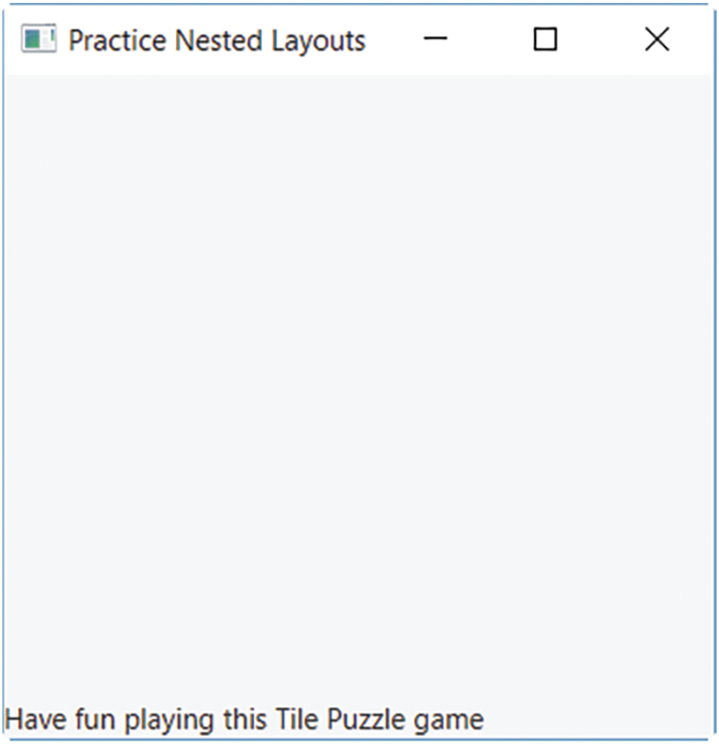
Figures 17.30 and **17.31** show the application running as the user first hovers over the “Cube” button and then clicks on it.

Figure 17.30 The “Cube” Button Changes Color as the Mouse Hovers Over It

Description

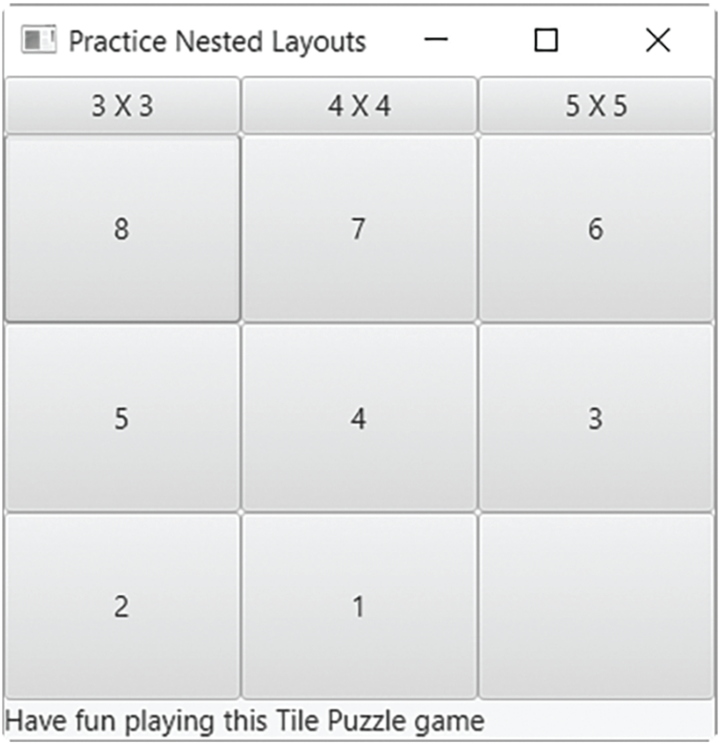
Figure 17.31 The User Entered 3 and Clicked on “Cube”

Description



Practice Nested Layouts

Have fun playing this Tile Puzzle game



Have fun playing this Tile Puzzle game

17.18 Programming Activity 2: Working with Layout Containers

In this Programming Activity, you will complete the implementation of a version of the Tile Puzzle game using a more complex GUI. As it stands now, the application compiles and runs, but it is missing a lot of code. **Figure 17.32** shows the window that will open when you run the application without adding your code. Once you have completed the five tasks of this Programming Activity, you should see the window in **Figure 17.33** when you run your program and click on the “3-by-3” button. When you click on one of the buttons labeled “3-by-3,” “4-by-4,” or “5-by-5,” the tile puzzle will reset to a grid of that size.

Figure 17.32 The Starting Window When Running the Prewritten Code

Description

Figure 17.33 The Starting Window When the Activity Is Completed

Description

In addition to the *TilePuzzle* class, we provide you with a prewritten *GameView* class, which encapsulates a View for the Tile Puzzle game. We have implemented the *GameView* class as a *GridPane* container, so you can add it to another layout container, such as a *BorderPane*. It has two important methods, shown in **Table 17.52**. Thus, your job in this Programming Activity is not to write the game code, but to organize components in a window.

The *GameView* Class API

Constructor

`GameView( int nSides )`

instantiates a tile puzzle View having an *nSides*-by-*nSides* grid

Return value	**Method name and argument list**
`void`	`setUpGame( int nSides )`

resets the grid as an *nSides*-by-*nSides* grid

Description

You need to edit the *NestedLayoutPractice* class, which *extends BorderPane*. Your job is to:

1. Declare an HBox named *top* and three *Buttons* that will be added to the *top* position of the *BorderPane*.
2. Set the layout containers for the *center* and *top* positions.
3. Add the *top* and the *gameView* layout containers to the

BorderPane.

4. Code an appropriate *private* listener class.
5. Instantiate the listener and register it on the appropriate components.

Instructions

Copy the source files in the Programming Activity 2 folder for this chapter to a folder on your computer.

1. Write the code to declare the needed instance variables. Load the *NestedLayoutPractice.java* source file and search for five asterisks in a row (*****). This will position you at the instance variables declaration.

```
// ***** Task 1: declare an HBox named top
// also declare three Button instance variables
// that will be added to the HBox top.
// These buttons will determine the grid size of the game:
// 3-by-3, 4-by-4, or 5-by-5

// task 1 ends here
```

   Description

2. Next, write the code to set the layout manager of the window and add the component *gameView* in the center position of the window. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you inside the constructor.

```
// ***** Task 2: student code starts here
// instantiate the GameView object

// add gameView to the center of this BorderPane

// task 2 ends here
```

   Description

3. Next, write the code to instantiate the *HBox* top component, instantiate the buttons from task 1, add them to *top*, and

finally add *top* at the top position of our *BorderPane*. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you inside the constructor.

```
// ***** Task 3: Student code restarts here
// instantiate the HBox component named top
// instantiate the Buttons that determine the grid size

// add the buttons to HBox top
// make them take all the available space
// add HBox top to this BorderPane as its top component

// task 3 ends here
```

Description

4. Next, write the code for the *private* inner class that implements the appropriate listener. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you between the constructor and the end of the class.

```
// ***** Task 4: Student code restarts here
// create a private inner class that implements EventHandler
// your method should identify which of the 3 buttons
//    was the source of the event
// depending on which button was pressed,
//    call the setUpGame method of the GameView class
//    with arguments 3, 4, or 5
// the API of that method is:
//   public void setUpGame( int nSides )

// task 4 ends here
```

Description

5. Next, write the code to declare and instantiate a listener, and register it on the appropriate components. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you inside the constructor.

```
// ***** Task 5: Student code restarts here
// Note: search for and complete Task 4 before performing this task
// declare and instantiate an EventHandler

// register the handler on the 3 buttons
// that you declared in Task 1

// task 5 ends here
```

Description

After completing each task, compile your code.

When you have finished writing all of the code, compile the source code and run *NestedLayoutPracticeApplication*. Try clicking on the three buttons that you added.

DISCUSSION QUESTIONS

1. Identify the various layout containers you used and the screen positions they occupy.
2. Explain why the left and right positions are not shown on the window.

© Lois GoBe/Shutterstock

CHAPTER REVIEW

17.19 Chapter Summary

- A Graphical User Interface (GUI) allows the user to enter data and initiate actions for an application by entering text into boxes; pressing buttons; moving the thumb of a slider; or selecting items from a list, a set of radio buttons, or checkboxes.
- Applications with GUIs are usually easier to learn and use because the interface is familiar to the user.
- JavaFX is a set of classes included with Java SE for creating GUIs.
- The top-level structure in a JavaFX application is the stage, which corresponds to a window. A stage can have one or more scenes, which are top-level containers for nodes that make up the window contents. A node can be a user interface control, such as a button or drop-down list; a layout container; an image or other media; a graphical shape; a web browser; a chart; or a group.
- JavaFX applications can be built in several ways. If we know which controls our interface needs and how they should be arranged, we can specify the layout, controls, and their properties using FXML, a scripting language based on XML. For more complicated GUIs or for dynamic GUIs where the number or type of control is determined at runtime, we can control the number, type, properties, and

positioning of controls programmatically.

- A JavaFX application extends the *Application* class and, at minimum, implements the *start* method.
- Some JavaFX GUI controls are *Label*, *TextField*, *Button*, *RadioButton*, *CheckBox*, *ComboBox*, and *Slider*. These controls inherit from the *Node* class.
- Controls can be arranged in a window using layout containers, such as *HBox*, *VBox*, *BorderPane*, *GridPane*, or *StackPane*. Layout containers can be nested.
- GUI applications use an event-driven model, where the user determines which functions are performed by interacting with the application controls, consequently firing events. To handle an event, we register our interest in being notified of the event and provide code: an event handler or listener, to be executed when the event occurs.
- JavaFX supports and encourages the Model-View-Controller architecture in GUI applications. The Model manages the data of the application and its state. The View presents the user interface. The Controller handles events generated by the user and communicates those changes to the Model, which updates its state accordingly and communicates any changes back to the Controller. The Controller then updates the View to reflect those changes.
- FXML can be used to define not only the application's layout containers, GUI controls, and their properties, but also the controller class and the method to be executed when an event fires.
- An *ActionEvent* object is created when the user clicks a button, selects an item from a list or a menu, or presses the *Enter* key in a *TextField*. The *getSource* method of the *ActionEvent* object returns a reference to the control that fired the event.
- To make *RadioButtons* mutually exclusive, we define a *ToggleGroup* and then set each *RadioButton*'s *toggleGroup* property accordingly.
- A Controller can have an *initialize* method, which is called after the scene graph has been created. The *initialize* method can be used to add nodes and set properties that

could not be fully defined in the FXML file. In the *initialize* method, we also can retrieve initial values from the Model and update the View accordingly so that the View reflects the initial state of the Model when the application starts.

- We put *ComboBox* items into an *ObservableList* and use the *SingleSectionModel* to manage the selection of items.
- The *Slider* control is capable of displaying a set of continuous values along a horizontal or vertical line called a track. The user “slides” the knob, called the thumb, along the track to select a value. We can set properties of a *Slider* to display tick marks and tick values.
- An event handler for a *Slider* control needs to implement the *ChangeListener<T>* interface.
- Defining our application’s GUI components, properties, and event handlers programmatically is useful for dynamic GUIs where the number or type of controls is not known until run time, or when we have an array of controls that are handled similarly.
- A *VBox* is a layout container that arranges its components vertically.
- An *HBox* is a layout container that arranges its components horizontally.
- A *GridPane* can be visualized as a table made up of cells in rows and columns. Each cell can contain one component. These cells can have different sizes, which we specify using row and column constraints.
- The *Alert* class enables us to construct and display dialog boxes.
- A *BorderPane* layout container organizes its nodes into five positions—*top*, *bottom*, *left*, *right*, and *center*—with each position holding one node at most.
- Lambda expressions can be used to simplify the definition of an event handler as an anonymous class that *implements* a functional interface, that is, an interface that requires only one method to be implemented.
- A lambda expression contains (1) a comma-separated list of parameters enclosed in parentheses—the data types of the parameters may be omitted, and the parentheses can

also be omitted if there is only one parameter; (2) the arrow token, ->; and (3) a method body, which can be a single expression or a block enclosed in curly braces. If the body of the method consists of a single expression, then the JVM evaluates the expression and returns its value. We can also use a return statement, but that requires curly braces.

- JavaFX includes many classes that allow us to animate a node on a scene. *FadeTransition*, *RotateTransition*, *TranslateTransition*, and *ScaleTransition* are examples of such classes.
- Each transition class typically provides methods to play the animation, set the number of types it repeats, play the animation in reverse, and execute some code after it finishes.
- A mouse activity generates a *MouseEvent*. A touch activity generates a *TouchEvent*.
- To handle a mouse or touch event, we implement the *EventHandler* interface and implement its *handle* method.
- The *getScene* method from the *Node* class allows us to get a reference to the *Scene* that a *Node* is in. The *getWindow* method from the *Scene* class allows us to get a reference to the *Stage* that the scene is in.
- JavaFX includes classes that make it easy to display various types of charts, including line charts, pie charts, and bar charts.
- The *ObservableList* interface is a list that can be linked to a GUI component that displays a list of values. Whenever the values in the list change, the GUI component is automatically updated to reflect those changes.
- *PieChart* and *BarChart* are examples of classes that can be bound to an *ObservableList*.

17.20 Exercises, Problems, and Projects

17.20.1 Multiple Choice Exercises

1. An example of a GUI component class is

   - *FXML*
   - *controller*
   - *TextField*
   - *Stage*

6. What are the primary uses of GUI components? (Check all that apply.)

   - Display information
   - Facilitate the coding of methods
   - Let the user control the program
   - Collect information from the user

11. In what package do you find the *Button*, *TextField*, and *ComboBox* classes?

    - *javafx.scene*
    - *javafx.scene.control*
    - *java.scene*
    - *java.awt*
    - *java.io*

17. V*Box* is a

    - Label
    - Layout container that arranges components vertically
    - Layout container that arranges components

horizontally
- Scene

22. The property of a *Label* element that specifies the text inside the label is

    - *Label*
    - *Word*
    - *Phrase*
    - *text*

27. The property that specifies the name of the Controller class for the View defined in the FXML document is

    - *control*
    - *fx:control*
    - *controller*
    - *fx:controller*

32. The property of a *Button* element that specifies the method called when the user clicks the button is

    - *action*
    - *press*
    - *onAction*
    - *onPress*

37. What attribute and annotation do we use with an instance variable of the Controller to reference a GUI component defined in an FXML document?

    - *id* and *FXML*
    - *id* and *@FXML*
    - *fx:id* and *FXML*
    - *fx:id* and *@FXML*

42. Assume that we have correctly defined the FXML attribute of a *Button* element that specifies the method to call when the user clicks on that button. What is the return type of that method?

    - *Button*
    - *void*
    - *boolean*
    - *onAction*

47. With JavaFX, a user interface must be defined using FXML; it cannot be defined programmatically.

    - True
    - False

50. We want to set up event handling programmatically when the user clicks on a button. What should the programmer do? (Check all that apply.)

    - Code a class that implements the *EventHandler<ActionEvent>* interface.
    - Declare and instantiate an object reference (a listener) of the class above.
    - Call the *handle* method.
    - Register the listener on the button.

55. Assuming everything has been coded correctly in the previous question, what happens when the user clicks a button?

    - The *handle* method executes.
    - The *Button* constructor executes.
    - The *start* method executes.

59. We want to build a class that implements an interface that

listens to key events. What interface should we implement?

- *EventHandler<ActionEvent>*
- *EventHandler<KeyEvent>*
- *EventHandler<MouseEvent>*
- *KeyHandler<Event>*

64. We are designing a GUI programmatically with three buttons; a different action will be taken depending on which button the user clicks. We want to code only one *private* class implementing the *EventHandler<ActionEvent>* interface. Inside the *handle* method, which method do we call to determine which button was clicked?

    - *getButton*
    - *getSource*
    - *getOrigin*

68. *NewLayout* is a layout container.

- True
- False

71. What is the maximum number of top-level controls that *a BorderPane* can manage?

    - 2
    - 3
    - 4
    - 5
    - 6

77. FXML elements can be nested.

    - True
    - False

80. Which one is not a transition class?

    - *FadeTransition*
    - *SequentialTransition*
    - *ParallelTransition*
    - *Scale*

85. What class encapsulates a touch event?

    - *Touch*
    - *TouchEvent*
    - *EventTouch*

89. What does the *getData* method of *XYChart.Series* return?

    - An *XYChart.Data* items
    - An *ObservableList* of *XYChart.Data* items
    - A *double*
    - A *Series*

17.20.2 Reading and Understanding Code

For Questions 21 to 25, consider the following FXML file representing the View for the application:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<HBox fx:controller="MyController"
       xmlns:fx="http://javafx.com/fxml"
       alignment="center" spacing="10" >
  <Button fx:id="button1" text="Button 1"
          onAction="#go" />
  <Button fx:id="button2" text="Button 2"
          onAction="#go" />
  <Label fx:id="result" />
  <Button fx:id="button3" text="Button 3"
          onAction="#go" />
  <Button fx:id="button4" text="Button 4"
          onAction="#go" />
</HBox>
```

Description

1. How many buttons will be displayed in the window?
2. How are the buttons and the label organized in the window?
3. What class should we code in order to process clicks on the buttons by the user?
4. What method will execute when the user clicks on one of the buttons?
5. What is the return type of that method?

   For Questions 26 through 29, consider the following code (and assume that the FXML file defines the text for cb1, cb2, and cb3 as “Choice 1,” “Choice 2,” and “Choice 3”):

```
 /*  Controller class
 *   Anderson, Franceschi
 */
 import javafx.event.ActionEvent;
 import javafx.fxml.FXML;
 import javafx.scene.control.*;

 public class Controller
  {
   @FXML private CheckBox cb1;
   @FXML private CheckBox cb2;
   @FXML private CheckBox cb3;
   @FXML private Label label;
   private int read, write, execute;
   @FXML protected void mix( ActionEvent event )
    {
     CheckBox cb = ( CheckBox ) event.getSource( );
     if ( cb == cb1 )
       read = ( cb.isSelected( ) ? 4 : 0 );
     else if ( cb == cb2 )
       write = ( cb.isSelected( ) ? 2 : 0 );
     else if ( cb == cb3 )
       execute = ( cb.isSelected( ) ? 1 : 0 );
     int mode = read + write + execute;

     label.setText( "mode: " + mode );
  }
}
```

Description

6. What happens when the user checks “Choice 1” only?
7. What happens when the user checks “Choice 1” and then checks “Choice 2”?
8. What happens when the user checks “Choice 1,” then “Choice 2,” and then “Choice 3”?
9. What happens when the user checks “Choice 3,” then “Choice 2,” and then “Choice 2” again?

For Questions 30 through 32, consider the following code (and assume that the class extending *Application* exists and is correctly coded):

```
import javafx.event.*;
import javafx.scene.control.Button;
import javafx.scene.layout.*;
public class BoardView extends GridPane
{
  private Button [ ][ ] buttons;

  public BoardView( )
  {
    super( );

    ColumnConstraints col = new ColumnConstraints( );
    col.setPercentWidth( 25 );
    RowConstraints row = new RowConstraints( );
    row.setPercentHeight( 20 );

    for ( int i = 0; i < 5; i++ )
      getRowConstraints( ).add( row );
    for ( int j = 0; j < 4; j++ )
      getColumnConstraints( ).add( col );
    buttons = new Button[5][4];
    ButtonHandler bh = new ButtonHandler( );
    for ( int i = 0; i < 5; i++ )
    {
      for ( int j = 0; j < 4; j++ )
      {
        // instantiate the buttons
        buttons[i][j] = new Button( );
        buttons[i][j].setMaxWidth( Double.MAX_VALUE );
        buttons[i][j].setMaxHeight( Double.MAX_VALUE );
        add( buttons[i][j], j, i );
        buttons[i][j].setOnAction( bh );
      }
    }
  }
  private class ButtonHandler implements EventHandler<ActionEvent>
  {
    public void handle(  ActionEvent event )
    {
      for ( int i = 0; i < buttons.length; i++ )
        for ( int j = 0; j < buttons[i].length; j++ )
          if ( event.getSource( ) == buttons[i][j] )
          buttons[i][j].setText( "" + String.valueOf ( i + j ) );
  }
 }
}
```

Description

10. How many rows and columns are in the grid?

11. What happens when the user clicks on the *Button* located at the top left of the grid?

12. What happens when the user clicks on the *Button* located at the bottom right of the grid?

17.20.3 Fill In the Code

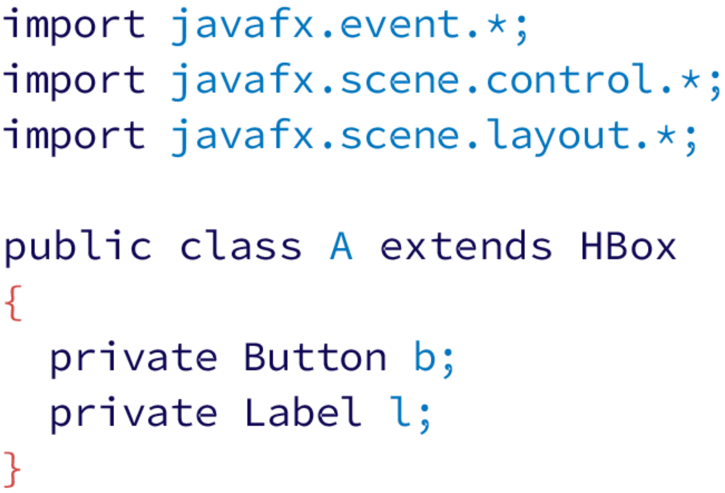
For Questions 33 through 35, consider the following FXML document representing a GUI:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<BorderPane fx:controller="MyController"
            xmlns:fx="http://javafx.com/fxml"  >
  <top>
    <HBox alignment="center">
      <Button fx:id="button1" text="INSERT" />
    </HBox>
  </top>

  <left>
    <HBox alignment="center">
      <Button fx:id="button2" text="UPDATE" onAction="#go"/>
    </HBox>
  </left>

  <!-- answer to questions 33 to 35 go here -->



</BorderPane>
```

Description

1. Add a *VBox* element that contains a button at the *right* position within the *BorderPane*. The text of the button should say DELETE; when the user clicks on it, the *test* method of the *MyController* class should execute. The id of the button should be *button3*.
2. Add an *HBox* element that contains a label at the *center* position of the *BorderPane*. The text of the label should be SELECT. The id of the label should be *label1*.
3. Add a *VBox* element that contains a button at the bottom position within the *BorderPane*. The text of the button should be CREATE; when the user clicks on it, the *table* method of the *MyController* class should execute. The id of the button should be *button4.*

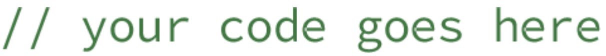
For Questions 36 through 40, consider the following class:



```
import javafx.event.*;
import javafx.scene.control.*;
import javafx.scene.layout.*;

public class A extends HBox
{
  private Button b;
  private Label l;
}
```

Description

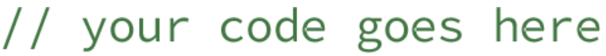
1. Inside the constructor, this code instantiates the button *b* with the text “Button.”



```
// your code goes here
```

Description

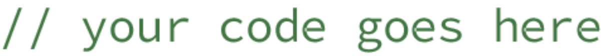
2. Inside the constructor, this code instantiates the label *l* with the text “Hello.”



```
// your code goes here
```

Description

3. Inside the constructor, this code adds *b* and *l* to this *HBox* so that *b* is on the right and *l* is on the left:



```
// your code goes here
```

Description

4. Inside the constructor, this code registers the listener *mh*

on the button *b*:

```
// the MyHandler class is a private class implementing EventHandler
MyHandler mh = new MyHandler( );
// your code goes here
```

Description

5. Inside the *handle* method of a *private* inner class implementing the *EventHandler* interface, this code changes the text of *l* to “Button clicked” if the button *b* was clicked:

```
public void handle( ActionEvent ae )
{
 // your code goes here
}
```

Description

For Questions 41 through 46, consider the following class:

```
import javafx.scene.control.*;
import javafx.scene.layout.*;

public class B extends BorderPane
{
  private VBox left;
  private HBox top;
  private Button [ ] buttons; // length 4
  private TextField [ ] textfields; // length 3
  private Label label1;
  private Label label2;

}
```

Description

Also, assume that none of the instance variables have been instantiated and you are coding inside the constructor.

1. This code instantiates *top* and *left*.

```
// your code goes here
```

Description

2. This code instantiates the text fields with text *TF0*, *TF1*, and *TF2*, and it places them in that order inside *top*. They should fill the whole available width of *top*.

```
// your code goes here
```

Description

3. This code instantiates the buttons with text *Button 0, Button 1*, *Button 2*, and *Button 3*, and it places them in that order inside *left*. They should fill the whole available height of *left*.

```
// your code goes here
```

Description

4. This code adds *left* and *top* at the left and top positions of the *BorderPane*, respectively.

```
// your code goes here
```

Description

5. This code instantiates *labe11* and *label2* with the text *CENTER* and *BOTTOM*, respectively.

```
// your code goes here
```

Description

6. This code adds *label1* and *label2* at the center and bottom positions within the *BorderPane*, respectively.

```
// your code goes here
```

Description

7. Replace the anonymous class definition for a *Button* event handler with a lambda expression.

```
quit.setOnAction( new EventHandler<ActionEvent>( )
                 {
                   @Override
                   public void handle( ActionEvent event )
                   {
                     System.exit( 0 );
                   }
                 }
);
```

   Description

8. We have a reference named *myCanvas* to a *Canvas*. Define a rotation animation that will take place on *myCanvas* as follows: it should last 2.5 seconds, repeat five times, and go from a 45-degree angle to a 135-degree angle.
9. A sound file named *music.wav* is located in the current folder. Write code to play the sound three times in a row.
10. We have a reference named *myEvent* to a touch event that just happened. Retrieve and output the *x* and *y* coordinates of where the touch event happened.
11. We have already defined an *ArrayList* of *PieChart.Data* named *myList*. Create an *ObservableList* with it, and create a *PieChart* using that *ObservableList*.

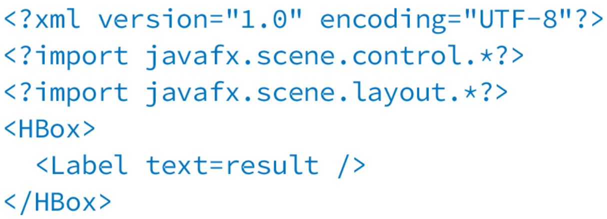
17.20.4 Identifying Errors in Code

1. Where is the error in this code sequence?

```
<?xml version="1.0" encoding="UTF-8"?>
<HBox xmlns:fx="http://javafx.com/fxml"
      alignment="center" spacing="10" >
  <Button text="Button 1" />
</HBox>
```

   Description

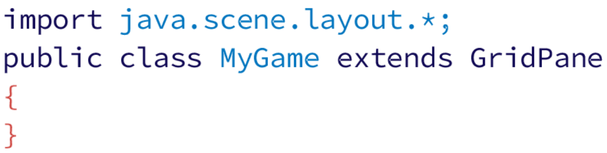
2. Where is the error in this code sequence?



```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<HBox>
  <Label text=result />
</HBox>
```

Description

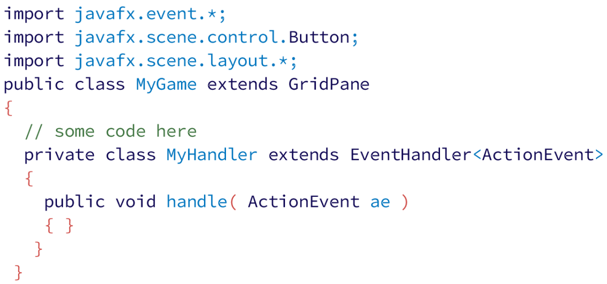
3. Where is the error in this code sequence?



```
import java.scene.layout.*;
public class MyGame extends GridPane
{
}
```

Description

4. Where is the error in this code sequence?



```
import javafx.event.*;
import javafx.scene.control.Button;
import javafx.scene.layout.*;
public class MyGame extends GridPane
{
  // some code here
  private class MyHandler extends EventHandler<ActionEvent>
  {
    public void handle( ActionEvent ae )
    { }
   }
 }
```

Description

5. Where is the error in this code sequence?

```
import javafx.event.*;
import javafx.scene.control.Button;
import javafx.scene.layout.*;
public class MyGame extends GridPane
{
  // some code here
  private class MyHandler implements EventHandler
  {
    public void handle( ActionEvent ae )
    { }
  }
}
```

Description

6. Where is the error in this code sequence?

```
XYChart.Series<String, Number> series
  = new XYChart.Series<String, Number>( );
series.add( new XYChart.Data<String, Number>( "HI", 2.0 ) );
```

Description

7. Where is the error in this code sequence?

```
// bc is a BarChart and has been instantiated
ArrayList<XYChart.Series<String, Number>> series
  = new ArrayList<XYChart.Series<String, Number>>( );
// fill in series here
bc.addAll( series );
```

Description

17.20.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. You coded the following class:

```
import java.net.URL;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.layout.VBox;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Test59 extends Application
{
  @Override
  public void start( Stage stage )
  {
    try
    {
      URL url = getClass( ).getResource( "fxml_ex1.fxml" );
      VBox root = FXMLLoader.load( url );
      Scene scene = new Scene ( root, 300, 275 );
      stage.setTitle( "Test" );
      stage.setScene( scene );
      stage.show( );
    }
    catch ( Exception e )
    {
      System.out.println( e.getMessage( ) );
    }
   }

  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Description

The code compiles; when you run, the window does not open and you get the following message:

```
java.lang.NullPointerException: Location is required
```

Description

What do you think the problem is?

2. You coded the following FXML file, whose name is *fxml:ex60.fxml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<VBox xmlns:fx="http://javafx.com/fxml"
       alignment="center" spacing="20" >

  <Label text=Welcome />
  <Label text="FXML Test" />
</VBox>
```

Description

Assume that the *Application* class is correctly coded. When you run, the window does not open and you get a LoadException with this message:

```
javafx.fxml.LoadException:
fxml:ex60.fxml:6
```

Description

Explain what the problem is and how to fix it.

3. You coded the following FXML file *(fxml_ex61.fxml*) and *Controller61* class:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox fx:controller="Controller61"
       xmlns:fx="http://javafx.com/fxml"
       alignment="center" spacing="10" >
  <Button fx:id="button" text="GO"
              onAction="#go" />
</VBox>
```

```
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.*;

public class Controller61
{
  protected void go( ActionEvent event )
  {
    System.out.println( "Inside go" );
  }
}
```

Description

Assume that the *Application* class is correctly coded. The code compiles but when you run, the window does not open and you get a message starting with:

```
javafx.fxml.LoadException: Error resolving onAction='#go',
either the event handler is not in the Namespace or there is
an error in the script
```

Description

Explain what the problem is and how to fix it.

4. You coded the following in the file *Test62.java*:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
public class Test62 extends Application
{
 public void start( Stage stage )
 {
  Ex62 root = new Ex62( );
  Scene scene = new Scene( root, 300, 275 );
  stage.setTitle( "Test" );
  stage.setScene( scene );
 }
 public static void main( String [ ] args )
 {
  launch( args );
 }
}
```

Description

Assume that the *Ex62* class is correctly coded. The code compiles and runs, but the window does not show. Explain what the problem is and how to fix it.

5. You coded the following in the *Ex63.java* file:

```
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.event.*;
public class Ex63 extends VBox
{
  private Button button;
  private Label label;

  public Ex63( )
  {
    super( );
    label = new Label( "HI" );
    button = new Button( "GO" );
    getChildren( ).add( label );
    getChildren( ).add( button );
  }
  private class ButtonHandler implements EventHandler<ActionEvent>
  {
    public void handle( ActionEvent ae )
    {
      label.setText( "Hello" );
    }
  }
}
```

Description

Assume that the *Application* class has been correctly coded. The code compiles and runs. However, when you click the button, the text in the label does not change. Explain what the problem is and how to fix it.

17.20.6 Write a Short Program

1. Write a program that displays a text field and two buttons labeled “uppercase” and “lowercase.” When the user clicks on the uppercase button, the text changes to uppercase; when the user clicks on the lowercase button, the text changes to lowercase. Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.
2. Write a program with two radio buttons and a text field. When the user clicks on one radio button, the text changes to lowercase; when the user clicks on the other radio button, the text changes to uppercase. Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.
3. Same as 65, except create the GUI programmatically, not using FXML.
4. Write a program with three checkboxes that allows a user to select toppings for a pizza order: extra cheese, sausage, and anchovies. As the user selects/deselects each checkbox, display the current order in a label.
5. Write a program that simulates a multiple choice question of your choice; the question should be a “check all that apply” type of question. There should be at least four possible answers, each using a checkbox. When the user selects any checkbox, your program should process the user’s answer and show whether the answer is true (all checkbox selections are correct) or false (at least one

checkbox selection is incorrect) in a label. Use FXML for the GUI and to set up event handling; include a Model.

6. Same as 68, but include a button to process the answer; do not process the answer when the user selects checkboxes.
7. Write a program that simulates a guessing game in a GUI program. Ask the user for a number between 1 and 6 in a text field, and then roll a die randomly and indicate whether or not the user won. Write the program in such a way that any invalid user input (i.e., not an integer between 1 and 6) is rejected and the user is asked again for input. Use FXML for the layout but generate and handle the checkboxes programmatically; include a Model. Be sure that the initial state of the View matches the initial state of the Model.
8. Write a program that simulates a guessing game in a GUI program. Generate a secret random number between 1 and 100; that number is hidden from the user. Ask the user to guess a number between 1 and 100 in a text field, and then tell the user whether the number is too high, too low, or is the correct number. Let the user continue to guess until the correct number is guessed. Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.
9. Modify the *ComboBox* example in this chapter to use *ListView* instead.

10. Write a program that displays a 4-by-6 grid of buttons, each with some unique text. One button is the “winning” button, which your model determines randomly. When the user clicks on the winning button, change its text to “Won.” If the user clicks on any other button, change its text to “No.” Use code for the GUI and to set up event handling; include a Model.
11. Same as Exercise 73 with the following additions: Keep track of how many times the user clicks on buttons. If the user has not won after five clicks, the text on the last button clicked should be changed to “Lost.” Once the user has lost or won, you should disable the game; that is, the buttons should no longer respond to clicks from the user.
12. Write a program that displays a combo box and a label. The combo box displays five U.S. states using two letters for each state (e.g., CA, MD). When the user selects a state from the combo box, the label is populated with the

full name of the state (e.g., if the user selects MD, the label is populated with Maryland). Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

13. Same as 75, except use code for the GUI instead of FXML.
14. Write a program that counts and displays how many times the user touches the screen and plays a short sound every time the user touches the screen. The count should be displayed and incremented each time the user touches the screen.
15. Write a program that plays the same greeting in three languages. Provide a button for each language. When the user presses a button, play the greeting in that language. Use code for the GUI and use lambda expressions to define the listener for each button.
16. Write a program that shows a ball bouncing on the ground. When the user touches the screen, the ball drops and bounces on the ground. The touch event should be disabled after that. The ball bounces down and up five times; each time it bounces up, it travels half the previous distance.
17. Write a program that shows a ball at the center of a *BorderPane*. Include a button at the bottom of the *BorderPane*. When the user clicks on the button, animate the ball so that it travels in a square. Disable the button while the animation is running and enable the button when the animation stops.

17.20.7 Programming Projects

1. Write a GUI-based tic-tac-toe game for two players. Use

code for the GUI and to set up event handling; include a Model.

2. Write a GUI-based program that analyzes a word. The user will type the word in a text field. Provide buttons for the following:

   - One button, when clicked, displays the length of the word.
   - Another button, when clicked, displays the number of vowels in the word.
   - Another button, when clicked, displays the number of uppercase letters in the word.

   Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

6. Write a GUI-based program that analyzes a soccer game. The user will type the names of two teams and the score of the game in four text fields. You should add appropriate labels and create buttons for the following:

   - One button, when clicked, displays which team won the game.
   - Another button, when clicked, displays the game score.
   - Another button, when clicked, displays by how many goals the winning team won.

   Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

10. Same as 83, except use code for the GUI, not FXML.

11. Write a GUI-based program that analyzes a round of golf. You will retrieve the data for 18 holes from a text file. Each line in the file will include the par for that hole (3, 4, or 5)

and your score for that hole. Your program should read the file and display a combo box listing the 18 holes. When the user selects a hole, the score for that hole should be displayed in a label. Provide buttons for the following:

- One button, when clicked, displays whether your overall score was over par, under par, or par.
- Another button, when clicked, displays the number of holes for which you made par.
- Another button, when clicked, displays how many birdies you scored. (A birdie on a hole is 1 under par.)

Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

15. Same as 85, except use code for the GUI, not FXML.
16. Write a GUI-based program that analyzes statistics for tennis players. You will retrieve the data from a text file. Each line in the file will list the name of a player, the player’s number of wins for the year, and the player’s number of losses for the year. Your program should read the file and display the list of players. When the user selects a player, the winning percentage of the player should be displayed in a label. Provide buttons for the scenarios that follow:

    - One button, when clicked, displays which player had the most wins for the year.
    - Another button, when clicked, displays which player had the highest winning percentage for the year.
    - Another button, when clicked, displays how many players had a winning record for the year.

    Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

20. Write a GUI-based program that simulates the selection of a basketball team. You will retrieve the data from a text file containing 10 lines. Each line will list the name of a player. Your program needs to read the file and display 10 checkboxes representing the 10 players. A text area will display the team, made up of the players being selected. A basketball team has five players. Your program should not allow the user to change his or her selection after the team has five players. Every time the user checks or unchecks a checkbox, the team in the text area should be updated accordingly. Provide buttons for the following:

    - One button, when clicked, displays how many players are currently on the team.
    - Another button, when clicked, displays how many players remain unselected.

    Use code for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

23. Write a GUI-based program that enables the user to choose a file containing an image. Your application then displays that image in a label. Your GUI should include a button and a label. When the user clicks on the button, open a file-choosing dialog box to enable the user to select a file. (Hint: look up the *FileChooser* class in the Java Class Library.) The dialog box should show the files in the current directory with an extension of either *jpg* or *gif*.

Use FXML for the GUI and to set up event handling; include a Model.

24. Write a GUI-based program that displays a team on a soccer field. You will retrieve data from a text file containing 11 lines. Each line will contain the name of a player. Your program should read the file and display the following window when it starts. (You can assume that the players in the file are not in any particular order.) Each cell is a button; when the user clicks on a button, the button replaces its text with the name of the player.

Left wing (11)	Striker (9)	Right wing (7)	
Left midfielder (6)	Midfielder (10)	Right midfielder (8)	
Left defender (3)	Stopper (4)	Sweeper (5)	Right defender (2)
	Goalie (1)		

Use FXML for the GUI and to set up event handling; include a Model.

25. Write a program that displays a color on canvas and allows

the user to change the color by changing its red, green, and blue components. Provide three sliders, with each slider representing a coefficient between 0 and 1 for the red, green, and blue amount, respectively. As the user manipulates each slider, change the color displayed on the canvas. Hint: use the *fillRect* method to create a rectangle the same size as the canvas.

Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

26. Same as 91, but create the GUI programmatically instead.
27. Write a GUI-based program that includes three sliders and a label. The sliders are used to define the red, green, and blue components of a color that you should use for the background color of the label. Each slider represents a coefficient between 0 and 255 for the red, green, and blue amount, respectively. As the user moves a slider, the background color of the label changes.

    Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.
28. Same as 93, but create the GUI programmatically instead.
29. Write a GUI-based program that generates a UNIX permission command; the UNIX permission command format is:

    ```
    chmod xyz filename
    ```

    Description

    where *x*, *y,* and *z* have values between 0 and 7.

    Provide the following:

    - Three combo boxes for the permission level for all, the group, and the owner of a file. In each combo box, the

  user can choose the permission level, a number between 0 and 7.
- One text field, where the user enters the name of a file.
- A label that displays the permission command for that file based on the values of the three combo boxes. Every time the user interacts with one of the combo boxes, the label should be updated.

Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

33. Same as 95, except that you should update the label every time the user updates the name of the file.
34. Same as 95, but create the GUI programmatically instead of with FXML.
35. Write a GUI-based program that simulates entering the destination in a car's GPS system. The user can enter a destination by typing it in a text field or by choosing a destination from a drop-down list of previous destinations. The list of previous destinations is sorted as follows: the most recent destination is at the top and the least recent is at the bottom of the list. If the user enters a destination in a text field, the user needs to click a button to validate it. A label displays the destination selected. When a destination is either entered or selected by the user, the list of previous destinations should be updated; the current destination goes to the top of the list. There should not be any duplicate destinations in the list. In this version, the list is empty when we start the program. Data are not persistent: Every time we start the program, the list of previous destinations is empty.

    Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

36. Same as 98, but the list of previous destinations should be persistent. It can be stored in a file and the drop-down list can be populated by the contents of the file.
37. Write a program that manages and displays a list of your friends' names and phone numbers. The list of names and phone numbers is displayed in a *ListView* component. Your program should allow the user to add and delete names and phone numbers one at a time. Furthermore, the data should be persistent and stored in a file. When the program starts, the contents of the file are displayed in the *ListView*. Every time a name and phone number are added or deleted, the file should be updated.

38. Write a drawing program. A *BorderPane* divides the screen into two parts: on the right, the drawing takes place. On the left are four buttons that enable the user to change the drawing color to any of three colors and to clear the drawing on the right side of the screen. The user draws by

touching and moving his or her finger on the screen. The drawing is made with the selected color.

39. Write a program that displays a pie chart on the left and its equivalent bar chart on the right. Both charts are dynamic and reflect four values that are input by the user at the bottom of the screen. You can use default values when the program starts.

17.20.8 Technical Writing

1. You are part of a team writing a complex program that includes a GUI. Our team is made up of programmers, as well as one artist and one HTML/XML developer who do not know programming but could learn some basic things quickly. We know that the GUI uses many different components and does not lend itself to using simple data structures like arrays. We know that the GUI is well defined and that the initial data in all its components are always the same. Would you define the GUI with FXML or programmatically? Discuss the pros and cons.

17.20.9 Group Project (for a group of 1, 2, or 3 students)

1. Design and code a program that simulates an auction. You should consider the following:
   A file contains a list of items to be auctioned. You can decide on the format of this file and its contents. For example, the file could look like this:

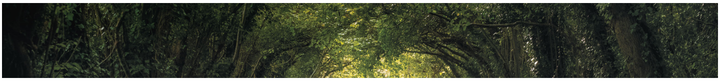
   ```
   Oldsmobile,oldsmobile.gif,100
   World Cup soccer ticket,soccerTickets.gif,50
   Trip for 2 to Rome,trip.gif,100
   ```

   Description

In the preceding file sample, each line represents an item as follows: The first field is the item’s description, the second field is the name of a file containing an image of the item, and the third field is the minimum bid. You can assume that each item’s description is unique.

Items are offered via an online-like auction. (You do not need to include any network programming; your program is a single-computer program.) Users of the program can choose which item to bid on from a list or combo box. Along with displaying the description of the item, your program should show a picture of the item and the current highest bid. (At the beginning, the current highest bid is the minimum bid.) Users bid on an item by selecting the item, typing a name (you can assume a different name), and entering a price for the item. Each time a bid is made, the item’s highest bid, displayed on the screen, should be updated if necessary.

Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.



© Lois GoBe/Shutterstock.

Appendix A

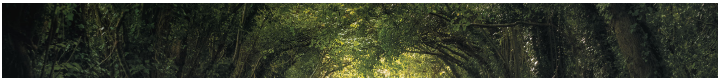
Java Reserved Words and Keywords

These words have contextual meaning for the Java language and cannot be used as identifiers.

abstract
assert
boolean
break
byte
case
catch
char
class
const
continue
default
do
double
else
enum
extends
false
final
finally
float
for
goto

if
implements
import
instanceof
int
interface
long
native
new
null
package
private
protected
public
record
return
short
static
strictfp
super
switch
synchronized
this
throw
throws
transient
true
try
void
volatile
while
yield

The words *true*, *false*, and *null* are literals. The remainder of the words are Java keywords, although *const* and *goto* are not currently used in the Java language.



© Lois GoBe/Shutterstock.

Appendix B

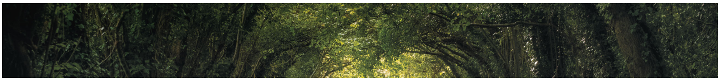
Operator Precedence

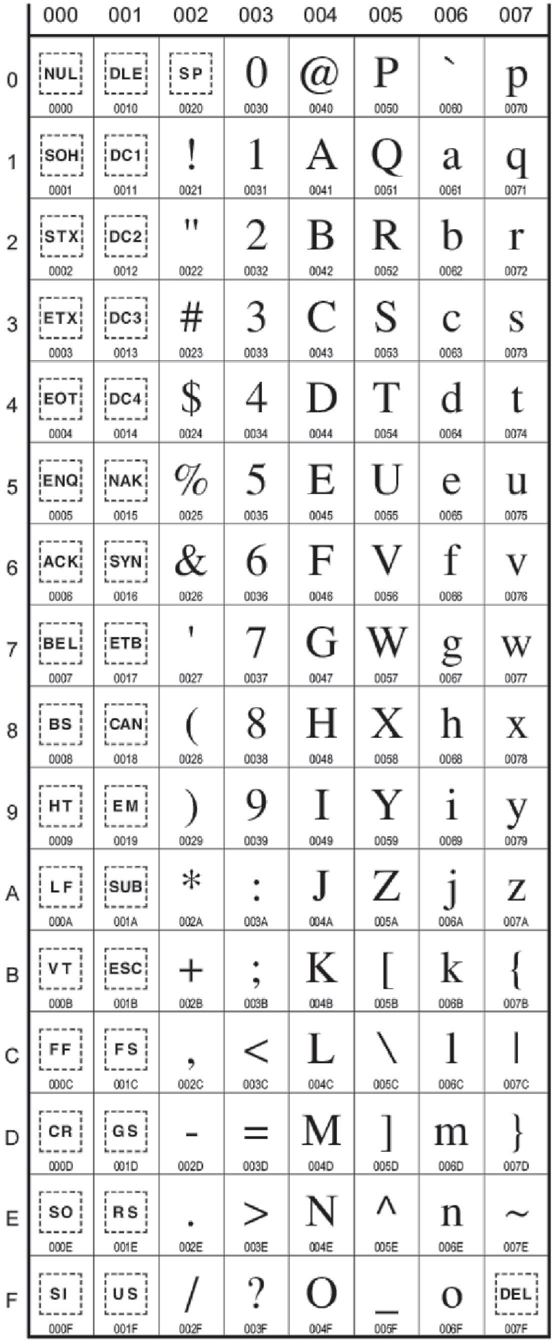
These rules of operator precedence are followed when expressions are evaluated. Operators in a higher level in the hierarchy—defined by their row position in the table—are evaluated before operators in a lower level. Thus, an expression in parentheses is evaluated before a shortcut postincrement is performed, and so on with the operators in each level. When two or more operators on the same level appear in an expression, the evaluation of the expression follows the corresponding rule for same-statement evaluation shown in the second column.

Operators	**Order of Same-Statement Evaluation**	**Operation**
( )	left to right	parentheses for explicit grouping
++—	right to left	shortcut postincrement and postdecrement
++— !	right to left	shortcut preincrement and predecrement, logical unary NOT
* / %	left to right	multiplication, division, modulus
+ -	left to right	addition or *String* concatenation, subtraction
< <= > >= instanceof	left to right	relational operators: less than, less than or equal to, greater than, greater than or equal

		to; instanceof
== !=	left to right	equality operators: equal to and not equal to
&&	left to right	logical AND
||	left to right	logical OR
?:	left to right	conditional operator
= += −= *= /= %=	right to left	assignment operator and shortcut assignment operators

Description





	000	001	002	003	004	005	006	007
0	NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	` 0060	p 0070
1	SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
2	STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
3	ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
4	EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
5	ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
6	ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
7	BEL 0007	ETB 0017	' 0027	7 0037	G 0047	W 0057	g 0067	w 0077
8	BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
9	HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
A	LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
B	VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
C	FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
D	CR 000D	GS 001D	- 002D	= 003D	M 004D	] 005D	m 006D	} 007D
E	SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
F	SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

© Lois GoBe/Shutterstock.

Appendix C

The Unicode Character Set

Java characters are encoded using the Unicode Character Set, which is designed to support international alphabets, punctuation, and mathematical and technical symbols. Each character is stored as 16 bits, so as many as 65,536 characters are supported.

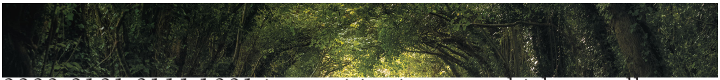
The American Standard Code for Information Interchange (ASCII) character set is supported by the first 128 Unicode characters from 0000 to 007F, which are called the controls and Basic Latin characters, as shown on the next page.

Any character from the Unicode set can be specified as a *char* literal in a Java program by using the following syntax: '\uNNNN' where NNNN are the four hexadecimal digits that specify the Unicode encoding for the character.

For more information on the Unicode character set, visit the Unicode Consortium's website: *www.unicode.org*.

Controls and Basic Latin Characters

Copyright © 1991–2010 Unicode, Inc. All rights reserved. Reproduced with permission of Unicode, Inc.



0000 0101 0111 1001 is a positive integer, which we call *a*.

1111 1111 1101 1010 is a negative integer, which we will call *b*.

$$a = 2^{10} + 2^{8} + 2^{6} + 2^{5} + 2^{4} + 2^{3} + 2^{0}$$
$$= 1,024 + 256 + 64 + 32 + 16 + 8 + 1$$
$$= 1,401$$

Two's complement of n = 2d − n

Two's complement of n = 2^d − 1 − n + 1

2^d − 1 1111 1111 1111 1111

− b 1111 1111 1101 1010

0000 0000 0010 0101

+ 1 1

two's complement of b 0000 0000 0010 0110

b 1111 1111 1101 1010

b inverted 0000 0000 0010 0101

+ 1 1

c 0000 0000 0010 0110

b 1111 1111 1101 1010

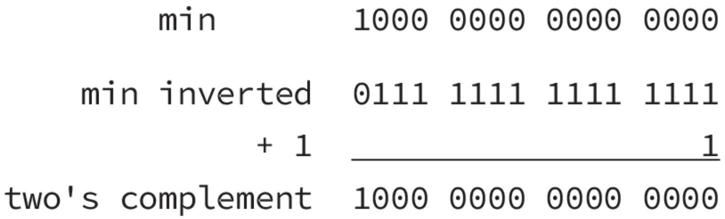
c 0000 0000 0010 0110

b + c 1 0000 0000 0000 0000

$$b = -(2^{5} + 2^{2} + 2^{1})$$
$$= -(32 + 4 + 2)$$
$$= -38$$

0111 1111 1111 1111

$$max = (2^{14} + 2^{13} + 2^{12} + 2^{11} + 2^{10} + 2^{9} + 2^{8} + 2^{7} + 2^{6} + 2^{5} + 2^{4} + 2^{3} + 2^{2} + 2^{1} + 2^{0})$$



```
           min       1000 0000 0000 0000

  min inverted       0111 1111 1111 1111
           + 1                         1
two's complement     1000 0000 0000 0000
```

© Lois GoBe/Shutterstock.

Appendix D

Representing Negative Integers

The industry standard method for representing negative integers is called **two’s complement**. Here is how it works:

For an integer represented using 16 bits, the leftmost bit is reserved for the sign bit. If the sign bit is 0, then the integer is positive; if the sign bit is 1, then the integer is negative.

For example, let’s consider two numbers, one positive and one negative.

Description

Using the methodology presented in the text for converting a binary number to a decimal number, we can convert the binary number, *a*, to its decimal equivalent. Hence, the value of *a* is calculated as follows:

Description

In contrast, *b*, the negative number, is represented in binary using the two’s complement method. The leftmost bit, which is the sign bit, is a 1, indicating that *b* is negative. To calculate the value of a negative number, we first calculate its two’s complement. The two’s complement of any binary number is another binary number, which, when added to the original number, will yield a sum consisting of all 0s and a carry bit of 1 at the end.

To calculate the two's complement of a binary number, *n*, subtract *n* from 2^d, where *d* is the number of binary digits in *n*. The following formula summarizes that rule:

Description

Knowing that $2^d - 1$ is always a binary number containing all 1s, we can simplify our calculations by first subtracting 1 from 2^d, then adding a 1 at the end.

Description

So to calculate the two's complement of *b*, which has 16 digits, we subtract *b* from a binary number consisting of 16 1s, then add 1, as shown here.

Description

Thus, the two's complement of *b*, which we will call *c*, is 0000 0000 0010 0110.

Another, simpler, way to calculate a two's complement is to invert each bit, then add 1. Inverting bits means to change all 0s to 1s and to change all 1s to 0s. Using this method, we get

Description

We can verify that the two's complement of *b* is correct by calculating the sum of *b* and *c*.

Description

Converting *c* to decimal will give us the value of our original number *b*, which, as we remember, is negative. We have

Description

Because a leftmost bit of 0 indicates that the number is positive, using 16 bits, the largest positive number (we will call it *max*) that we can represent is

Description

This is equivalent to $2^{15} - 1$, which is 32,768 – 1, or 32,767.

Using 16 bits, then, the smallest negative number (we will call it *min*) that we can represent is

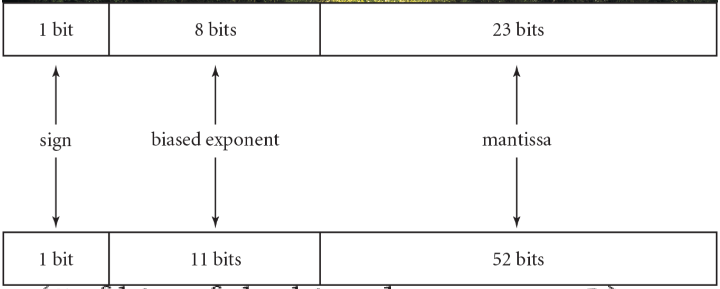
`1000 0000 0000 0000`

The two's complement of *min* is *min* itself. If we invert the bits and add 1, we get the same value we started with:

Description

and therefore *min* is -2^{15} or –32,768.

Thus, using 16 bits, we can represent integers between –32,768 and 32,767.


1 bit
8 bits
23 bits
sign
biased exponent
mantissa
1 bit
11 bits
52 bits

$$2^{(\#\text{ of bits of the biased exponent}-1)}-1$$

$$2^{(8-1)}-1=2^{7}-1=127$$

Real exponent	-127	-126	...	0	...	127	128
+Bias	127	127	...	127	...	127	127
Biased exponent	0	1	...	127	...	254	255

$$2^{(11-1)}-1=2^{10}-1=1023$$

$$(-1)^{\text{sign}}*(1+\text{significand})*2^{(\text{biased exponent}-\text{bias})}$$

$$=2^{7}+2^{2}+2^{1}+2^{0}$$

$$=128+4+2+1$$

$$=135$$

$$=(-1)^{0}*(1+.1101)*2^{(135-127)}$$

$$=1.1101*2^{8}$$

$$=11101\ 0000$$

$$= 2^8 + 2^7 + 2^6 + 2^4$$

$$= 256 + 128 + 64 + 16$$

$$= 464$$

$$= 1.1101 * 2^8$$

$$= (1 + 1 * \frac{1}{2}1 + 1 * \frac{1}{2}2 + 0 * \frac{1}{2}3 + 1 * \frac{1}{2}4) * 2^8$$

$$= (1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16}) * 2^8$$

$$= (1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16}) * 256$$

$$= 464$$

$$5 = 101$$

$$.375 = .25 + .125$$

$$= \frac{1}{4} + \frac{1}{8}$$

$$= \frac{1}{2}2 + \frac{1}{2}3$$

$$= 0 * \frac{1}{2}1 + 1 * \frac{1}{2}2 + 1 * \frac{1}{2}3$$

$$y = -101.11$$

$$= -1.01011 * 2^2$$

$$\text{Biased exponent} = 2 + 127$$

$$= 129$$

$$129 = 1000\ 0001$$

© Lois GoBe/Shutterstock.

Representing Floating-Point Numbers

IEEE 754, a specification accepted worldwide and used by the Java language, defines how to represent floating-point numbers in binary numbers. Single-precision floating-point numbers use 32 bits of memory, and double-precision floating-point numbers use 64 bits.
Here is how single- and double-precision floating-point numbers are represented:

Description

The leftmost bit stores the sign of the floating-point number; a 0 indicates a positive number, while a 1 indicates a negative number.
To represent the exponent of the number, which can be positive or negative, each representation stores a positive, biased exponent, calculated by adding a fixed bias, or scaling factor, to the real exponent of the number. The purpose of the bias is to be able to represent both extremely large and extremely small numbers. The bias is equal to

Description

Thus, for single precision, the bias is

Description

In single-precision, the 8-bit biased exponent can store 256 positive values (0 to 255). Thus, with a bias of 127, we can represent floating-point numbers with real exponents from –127 to 128, as shown here:

Description

Conversely, to find the real exponent from the biased exponent, we subtract the bias. For example, if the biased exponent is 150, then the real exponent is 150 − 127, which is 23. Similarly, if the biased exponent is 3, the actual exponent is 3 − 127, which is −124.

For double precision, the bias is

Description

A floating-point number is considered to be in the form

Description

By definition, the significand is of the form 0 followed by a dot followed by a string of 0s and 1s, for example, 0.1101. That string of 0s and 1s is known as the mantissa.

For example, if the significand is 0.1101, then the mantissa is 110100…0

As an example, let’s convert a single-precision binary number to a decimal floating-point number. We will convert the following single-precision IEEE 754 floating-point number:

0	10000111	11010000 … 0

The leftmost digit, 0, tells us that the number is positive. The biased exponent is 10000111, which converted to decimal, is

Description

The bias for single-precision floating-point numbers is 127, so the number is

Description

In decimal, the number is

Description

Given that .1 is ½1 or ½ in decimal, and .01 is ½2 or ¼, and .0001 is ½4 or 1/16 in decimal, we also could have calculated the number using this method:

Description

Now, let’s convert a decimal floating-point number into single-precision, binary format. Here, we will convert the number –5.375, which we’ll call *y*. First, without dealing with the sign yet, we convert the whole number portion (5) to binary, getting 101:

Description

Then we convert the fractional part to binary:

Description

Thus, .375 as represented in binary is .011.
Therefore, y can be represented in binary as

Description

We now can deduce the sign, the biased exponent, and the mantissa. The sign is 1 because the number is negative. The significand is 1.01011, and therefore the mantissa is 01011000…00. The exponent is 2, so the biased exponent is 129 (2 plus the bias for single-precision numbers, which is 127):

Description

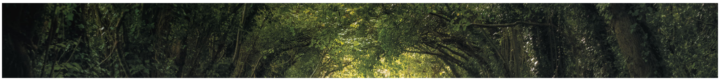
Converting 129 to binary, we get

Description

Therefore, the IEEE 754 single-precision value of the number *y* is

1	10000001	010110000…0

Description



© Lois GoBe/Shutterstock.

Appendix F

Solutions to Selected Exercises

1.7 Exercises, Problems, and Projects

1.7.1 Multiple Choice Exercises

1. Java
2. servers.
3. is a multiple of 4.
4. C
5. *javac Hello.java*

1.7.2 Converting Numbers

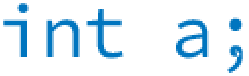
1. 0110 0001 1100
2. 0x15

1.7.3 General Questions

1. 3 billion
2. red = 51; green = 171; blue = 18
3. *javac*

2.6 Exercises, Problems, and Projects

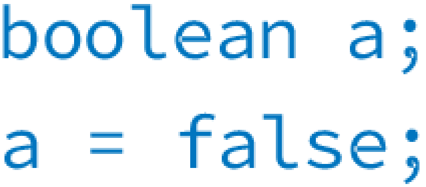
2.6.1 Multiple Choice Exercises

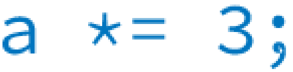
1. 
   ```
   int a;
   ```

2.6.2 Reading and Understanding Code

1. 12.5
2. 2.0
3. 4
4. 5
5. 2.4
6. 5
7. 0

2.6.3 Fill In the Code

1. 
   ```
   boolean a;
   a = false;
   ```

2.
   ```
   double avg = (double) ( a + b ) / 2;
   System.out.println( "The average is " + avg );
   ```
3. 
   ```
   a *= 3;
   ```

2.6.4 Identifying Errors in Code

1. Cannot assign a *double* to a *float* variable (possible loss of precision).
2. There should not be a space between – and =.

2.6.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. Cannot assign a *double* to an *int* variable (possible loss of precision). Change to:
   ```
   int a = 26;
   ```
2. =+ is different from += (shortcut operator). Here, *a* is assigned the value + 3. To add 3 to *a*, change the second statement to:
   ```
   a += 3;
   ```

3.18 Exercises, Problems, and Projects

3.18.1 Multiple Choice Exercises

1. `import`
2. `new`
3. It is a class method.
4. `double`
5. `Math.E;`

3.18.2 Reading and Understanding Code

1. hello
2. 3.141592653589793
3. 8

3.18.3 Fill In the Code

1. `System.out.println( s.length( ) );`

```
   System.out.print( "Welcome " );
   System.out.print( "to " );
   System.out.print( "Java " );
2. System.out.print( "Illuminated\n" );
   // code below assumes we have imported Scanner
   Scanner scan = new Scanner( System.in );
   System.out.print( "Enter two integers > " );
   int i = scan.nextInt( );
   int j = scan.nextInt( );
   int min = Math.min( i, j );
3. System.out.println( "min of " + i + " and " + j + " is " + min );
   // code below assumes we have imported Scanner
   Scanner scan = new Scanner( System.in );
   System.out.print( "Enter a double > " );
   double number = scan.nextDouble( );
   double square = Math.pow( number, 2 );
4. System.out.println( number + " squared = " + square );
```

3.18.4 Identifying Errors in Code

1. The *System* class begins with a capital S. It should be *System*, not system.
2. The *round* method of the *Math* class returns a *long*; a *long* cannot be assigned to a *short* variable due to a potential loss of precision.
3. The char *‘H’* cannot be assigned to the *String s*. The two data types are not compatible.

3.18.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. Java is case sensitive. The *Math* class needs to be spelled with an uppercase M.
2. In the output statement, we are just printing the value of *grade* without any formatting. To format *grade* as a percentage, the output statement should be:

```
System.out.println( "Your grade is " + percent.format( grade ) );
```

4.7 Exercises, Problems, and Projects

4.7.1 Multiple Choice Exercises

1. *javafx.scene.canvas*
2. true
3. the (*x, y*) coordinate of the upper-left corner of the rectangle we are drawing
4. 256

4.7.2 Reading and Understanding Code

1. 250 pixels

4.7.3 Fill In the Code

1.
   ```
   gc.setFill( Color.RED );
   gc.setStroke( Color.GREEN );
   ```

2.
   ```
   gc.fillRect( 50, 30, 50, 270 );
   ```

4.7.4 Identifying Errors in Code

1. There should be double quotes around the literal *Find a bug*, not single quotes. Single quotes are used for a *char*, not a *String*.
2. There is no *public color* instance variable in the *GraphicsContext* class. The *setFill* or *setStroke* method should be used to set the current color.

4.7.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. The rectangle is being drawn outside the bounds of the window. The largest visible x coordinate is 699.

5.14 Exercises, Problems, and Projects

5.14.1 Multiple Choice Exercises

☐ `a < b` `true`

☐ `a != b` `true`

☐ `a == 4` `false`

☐ `( b - a ) <= 1` `false`

☐ `Math.abs( a - b ) >= 2` `true`

☐ `( b % 2 == 1 )` `true`

1. ☐ `b <= 5` `true`
2. yes

☐ `a < b || b < 10` `no`

☐ `a != b && b < 10` `yes`

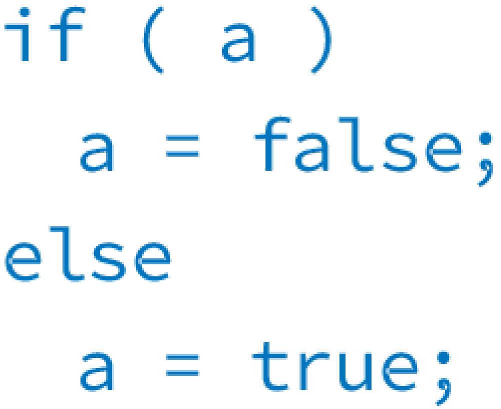
☐ `a == 4 || b < 10` `yes`

3. ☐ `a > b && b < 10` `no`

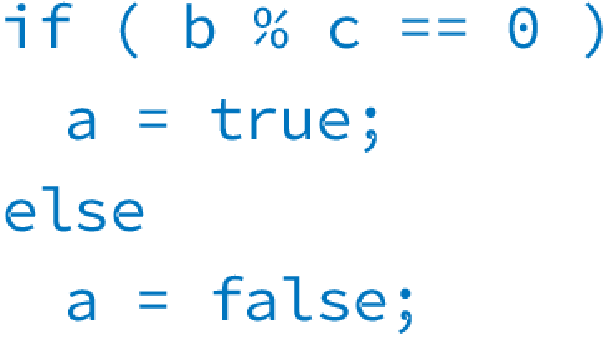
5.14.2 Reading and Understanding Code

1. *true*
2. 27 is divisible by 3
   End of sequence
3. Hello 3
   Hello 4
   Done
4. letter 2

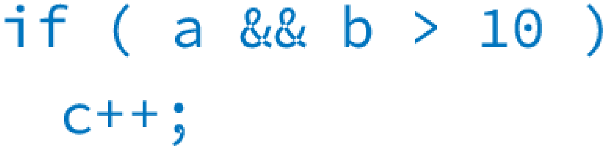
5.14.3 Fill In the Code

1. 

```
if ( a )
   a = false;
else
   a = true;
```

2. 

```
if ( b % c == 0 )
   a = true;
else
   a = false;
```

3. 

```
if ( a && b > 10 )
   c++;
```

5.14.4 Identifying Errors in Code

1. The && operator cannot be applied to two *int* operands (*a1* and *a2*).
2. We need a set of parentheses around *b1*.
3. There is no syntax error.

5.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. The expression `a = 31` evaluates to an *int*, 31. The *if* condition requires a *boolean* expression. To fix the problem, replace `a = 31` with `a == 31`.

6.14 Exercises, Problems, and Projects

6.14.1 Multiple Choice Exercises

1. The code runs forever.
2. true

6.14.2 Reading and Understanding Code

1. Enter an int > 3
   Enter an int > 5
   Hello
   Enter an int > –1
   Hello
2. *i* is 8 and *product* is 42
3. 3
4. *i* is 40 and *sum* is 60
5. 3
   3
   3
   3
   4

6.14.3 Fill In the Code

```
   System.out.print( "Enter an integer > " );
   int value = scan.nextInt( );
   while ( value != 20 )
   {
     if ( value >= start )
       System.out.println( value );
     System.out.print( "Enter an integer > " );
     value = scan.nextInt( );
1. }
   Scanner scan = new Scanner( System.in );
   System.out.print( "Enter a word > " );
   word = scan.next( );
   while ( ! word.equals( "end" ) )
   {
     sentence += word + " ";
     System.out.print( "Enter a word > " );
     word = scan.next( );
2. }
   Scanner scan = new Scanner( System.in );
   int sum = 0;
   System.out.println( "Enter an integer > " );
   int value = scan.nextInt( );
   while ( value != 0 && value != 100 )
   {
     sum += value;
     System.out.println( "Enter an integer > " );
     value = scan.nextInt( );
   }
3. System.out.println( "sum is " + sum );
```

6.14.4 Identifying Errors in Code

1. The variable *num* needs to be initialized after it is declared, and a priming read is needed.
2. The loop is infinite; *number* is always different from 5 or different from 7. The logical OR (`||`) should be changed to a logical AND (`&&`).

6.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. It is an infinite loop; *i* should be incremented, not decremented, inside the body of the *while* loop so that the loop eventually terminates.
2. In the *for* loop header, the loop initialization statement, the loop condition, and the loop update statement should be separated by semicolons (;), not commas(,).

7.19 Exercises, Problems, and Projects

7.19.1 Multiple Choice Exercises

1. The convention is to start with an uppercase letter.
2. true
3. can be basic data types, existing Java types, or user-defined types (from user-defined classes).
4. one parameter, of the same type as the corresponding field.
5. these fields do not need to be passed as parameters to the methods because the class methods have direct access to them.
6. All of the above.

7.19.2 Reading and Understanding Code

1. *double*
2. an instance method (keyword *static* not used)
3. `public static void foo3( double d )`

7.19.3 Fill In the Code

1.
   ```
   private int grade;
   private char letterGrade;
   ```
2.
   ```
   public TelevisionChannel( String newName, int newNumber,
                             boolean newCable )
   {
     name = newName;
     number = newNumber;
     cable = newCable;
   }
   ```
3.
   ```
   public String toString( )
   {
     return ( "name: " + name + "\tnumber: "
              + number + "\tcable: " + cable );
   }
   ```
4.
   ```
   public String typeOfChannel( )
   {
     if ( cable )
       return "cable";
     else
       return "network";
   }
   ```

7.19.4 Identifying Errors in Code

1. The *toString* method needs to return a *String*, not output data.
2. The method header is incorrect; it should be
   ```
   public double calcTax( )
   ```
3. There are two errors: The assignment operator = should not be used when declaring an *enum* set. And the *enum* constant objects should not be *String* literals but identifiers. The statement should be:
   ```
   enum Months { January, February, March };
   ```

7.19.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. The compiler understands that *Grade* is a method since its header says it returns a *char*. It looks as if it is intended to be a constructor so the keyword *char* should be deleted from the constructor header.
2. The constructor assigns the parameter *numberGrade* to itself, therefore not changing the value of the instance variable *numberGrade*, which by default is 0. The constructor could be recoded as follows:

```
public Grade( int newGrade )
{
  numberGrade = newGrade;
}
```

8.11 Exercises, Problems, and Projects

8.11.1 Multiple Choice Exercises:

1. `int [ ] a;` and `int a[ ];`
2. 0
3. `a.length`
4. false

8.11.2 Reading and Understanding Code

1. 48.3
2. 12

48
65

3. 14
4. It counts and returns how many elements in the argument array have the value 5.
5. It returns an array of *Strings* identical to the argument array except that the *Strings* are all in lowercase.

8.11.3 Fill In the Code

1.
```
if ( a[i] > 20 )
  System.out.println( a[i] );
```
2.
```
System.out.println( "a[" + i + "] = " + a[i] );
```
3.
```
if ( a.length < 2 )
  return false;
else if ( a[0].equals( a[1] ) )
  return true;
else
  return false;
```

8.11.4 Identifying Errors in Code

1. Index –1 is out of bounds; the statement `System.out.println( a[-1] );` will generate an *ArrayIndexOutOfBoundsException* at run time.
2. When declaring an array, the square brackets should be empty. Replace `a[3]` with `a[ ]`.
3. Although the code compiles, it outputs the hash code of the array a. To output the elements of the array, we need to loop through the array elements and output them one by one.

8.11.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. Index `a.length` is out of bounds; when i is equal to `a.length`, the expression `a[i]` will generate a run-time exception. Replace <= with < in the loop condition.

9.11 Exercises, Problems, and Projects

9.11.1 Multiple Choice Exercises

1. `int[ ][ ] a;` and `int a[ ][ ];`
2. false
3. `a[2].length`
4. true
5. *java.util*

9.11.2 Reading and Understanding Code

1. 3
2. Munich
   Stuttgart
   Berlin
   Bonn
3. Munich
   Berlin
   Ottawa
4. It counts and returns the number of elements in the parameter array *a*.

5. It returns an *int* array of the same length as the length of the parameter array *a*. Each element of the returned array stores the number of columns of the corresponding row in *a*.
6. 7 (at index 0) 45 (at index 1) 21 (at index 2)

9.11.3 Fill In the Code

1.
```
System.out.println( geo[0][5] );
for ( int i = 0; i < geo.length; i++ )
{
  for ( int j = 0; j < geo[i].length; j++ )
       System.out.println( geo[i][j] );
```
2.
```
}
int count = 0;
for ( int j = 0; j < a[1].length; j++ )
{
  if ( a[1][j] == 6 )
       count++;
}
```
3.
```
System.out.println( "# of 6s in the 2nd row: " + count );
```
4. This method returns the product of all the elements in an array.
```
public static int foo( int [ ][ ] a )
{
  int product = 1;
  for ( int i = 0; i < a.length; i++ )
  {
    for ( int j = 0; j < a[i].length; j++ )
    {
      product *= a[i][j];
    }
  }
  return product;
}
```
5.
```
System.out.println( languages.size( ) );
```

```
for ( String s : languages )
{
  if ( s.charAt( 0 ) == 'P' )
    System.out.println( s );
```

6. `}`

9.11.4 Identifying Errors in Code

1. Array dimension missing in `new double [ ][10]`

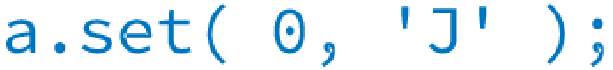
   Example of correct code: `double [ ][ ] a = new double [4][10];`
2. Cannot declare an *ArrayList* of a primitive data type; the type needs to be a class (for example: *Double*)
3. Correct syntax is `variable = expression`. Because `a.size( )` is not a variable, we cannot assign a value to it.

9.11.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. Other than `a[0][0]`, the first row is not taken into account because *i* is initialized to 1 in the outer loop. It should be `int i = 0;` not `int i = 1`.
2. Index 3 is out of bounds. There are only 3 elements in *a*; the last index is 2.
3. Because *ArrayList* elements begin at index 0, the statement

   ```
   a.set( 1, 'J' );
   ```

   sets the value of the second element of the *ArrayList*. To set the value of the first element, use this statement:



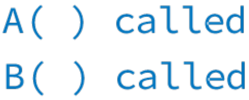
```
a.set( 0, 'J' );
```

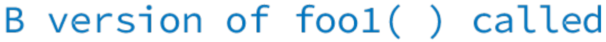
10.10 Exercises, Problems, and Projects

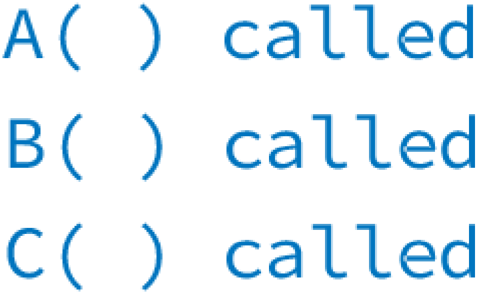
10.10.1 Multiple Choice Exercises

1. a class inheriting from another class.
2. *protected* and *public* instance variables and methods
3. You cannot instantiate an object from an *abstract* class.
4. the class must be declared *abstract*.

10.10.2 Reading and Understanding Code

1. *B* inherits from *A: name, price* (*foo2* and *foo3* are overridden)
   C inherits from *B*: *name, price, foo2*, and *foo3* (*foo1* is overridden)
   

   ```
   A( ) called
   B( ) called
   ```
2. 

   ```
   B version of foo1( ) called
   ```
3. 

   ```
   A( ) called
   B( ) called
   C( ) called
   ```

10.10.3 Fill In the Code

```
private char middle;
public G( String f, String n, char middle )
{
  super( f, n );
  this.middle = middle;
```

1. `}`
2. `public class K extends F implements I`

10.10.4 Identifying Errors in Code

1. There is no error. `new D( )` returns a *D* object reference. *D* inherits from *C*; therefore a *D* object reference “is a” *C* object reference. Thus, it can be assigned to *c2*. Although a *C* object cannot be instantiated, a subclass object reference can be assigned to a superclass reference.
2. The *foo* method does not have a method body; it must be declared *abstract*. The *K* class must be declared *abstract* as well.

10.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. The instance variable *n* of class *M* is private, and is not inherited by *P*. Therefore, *n* is not visible inside class *P*.

11.15 Exercises, Problems, and Projects

11.15.1 Multiple Choice Exercises

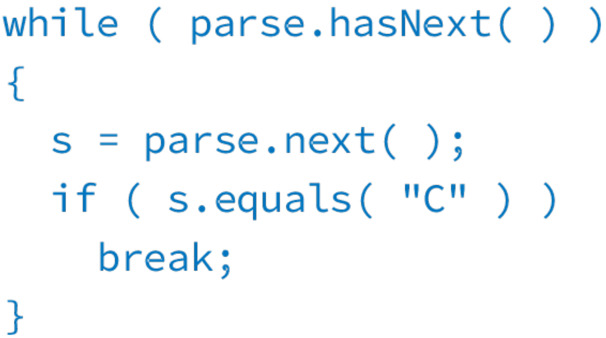
1. *try/catch* blocks enable programmers to attempt to recover from illegal situations and continue running the program.
2. false
3. the contents of the file, if any, will be deleted.

4. the *stream* method

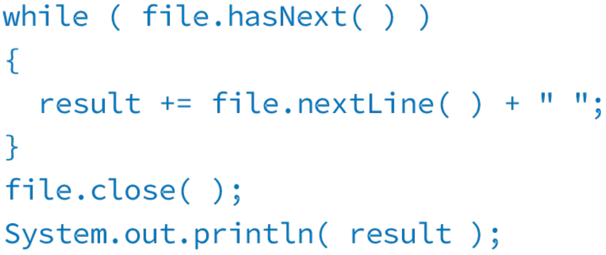
11.15.2 Reading and Understanding Code

1. ABCD
2. count = 3
3. Nice finish
4. CS1
   0
   1
   2
   3
   4

11.15.3 Fill In the Code

1. 

```
while ( parse.hasNext( ) )
{
  s = parse.next( );
  if ( s.equals( "C" ) )
    break;
}
```

2. 

```
while ( file.hasNext( ) )
{
  result += file.nextLine( ) + " ";
}
file.close( );
System.out.println( result );
```

```
average += grades[i];
. . .
average /= grades.length;
. . .
pw.println( average );
```

3. `pw.close( );`

11.15.4 Identifying Errors in Code

1. The *next* method returns a *String*; the return value cannot be assigned to an *int* variable. Use *nextInt* instead.

11.15.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. The *catch* block is missing; you need to add it after the *try* block as follows:

```
catch ( IOException ioe )
{
  ioe.printStackTrace( );
}
```

2. When we try to read 3.5 as an *int*, the method generates an *InputMismatchException* and we execute the *catch* block. To fix the problem, replace 3.5 with 3, for example, or use the *nextDouble* method and change *number* to a *double*.

12.12 Exercises, Problems, and Projects

12.12.1 Multiple Choice Exercises

1. may or may not be *static*.
2. calls itself.
3. a run-time error.

12.12.2 Reading and Understanding Code

1. 0
2. 3
3. There is no output
4. *foo3* outputs the parameter *String* in reverse
5. 64

12.12.3 Fill In the Code

1. `return foo( s.substring( 2, s.length( ) ) );`

```
if ( n >= 1000 )     // base case
  return n;
else                 // general case
```

2. `  return foo( n * n );`

12.12.4 Identifying Errors in Code

1. In the *else* clause, the *return* keyword is missing.

12.12.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. The base case is not coded properly; it needs to return a value, not make another recursive call. Instead of *return foo(0)*, you can code *return* 1.
2. In the general case, the method makes the recursive call with the original *String* less the last two characters as the argument. Therefore, there should be two base cases:

when the *String* has 0 characters (that is, an empty *String*) and when the *String* has one character. Assuming this method counts the number of characters in the *String* argument, we can add the following code after the first base case:

```
else if ( s.length( ) == 1 )
     return 1;
```

13.14 Exercises, Problems, and Projects

13.14.1 Multiple Choice Exercises

1. Linked lists are easily expanded.
2. (7, *Ajay*, *NFL*)
3. (7, *Ajay*, *NFL*) and (5, *Joe, Sonic*)
4. *front* = 3, stores 54; *back* = 0, stores 62
5. *front* = 7; *back* = 6; the list is empty

13.14.2 Reading and Understanding Code

1. If the list is not empty, it resets it to empty and returns *true*. Otherwise, it returns *false*.
2. It outputs all the *Player* objects in the list whose *game* field is *Diablo*.

13.14.3 Fill In the Code

1.

```
previous.getNext( ).setNext(
                    previous.getNext( ).getNext( ).getNext( ) );
```

```
public LLNode( char newGrade )
{
   grade = newGrade;
   next = null;
```

2.

```
}
```

```
public LLNode setGrade( char grade )
{
  this.grade = grade;
  return this;
}
public LLNode setNext( LLNode next )
{
  this.next = next;
  return this;
```

3. `}`

```
public int numberOfItems( )
{
  int count = 0;
  LLNode current = head;
  while ( current != null )
  {
    count++;
    current = current.getNext( );
  }
  return count;
```

4. `}`

13.14.4 Identifying Errors in Code

1. The *getID* method belongs to the *Player* class and cannot be called using *head*, a *PlayerNode* object reference. The code should be
   `return head.getPlayer( ).getID( );`
2. The number of items in the queue would never increase and we would always be able to insert into that queue, eventually overwriting items that are in the queue. This is a logic error. Furthermore, the queue would always be considered empty since the number of items always has

the value 0. We would never be able to delete an item from the queue.

13.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. There is no *getHead* method in the *PlayerLinkedList* class. In order to get a copy of the *Player* object stored at the first node of the list, we can code a method returning the *Player*.

14.8 Exercises, Problems, and Projects

14.8.1 Multiple Choice Exercises

1. $O(n^2)$
2. $O(n)$
3. $n * (n + 1) / 2$
4. $O(n^2)$

14.8.2 Compute the Running Time of a Method

1. $O(n)$
2. $O(\log n)$

15.10 Exercises, Problems, and Projects

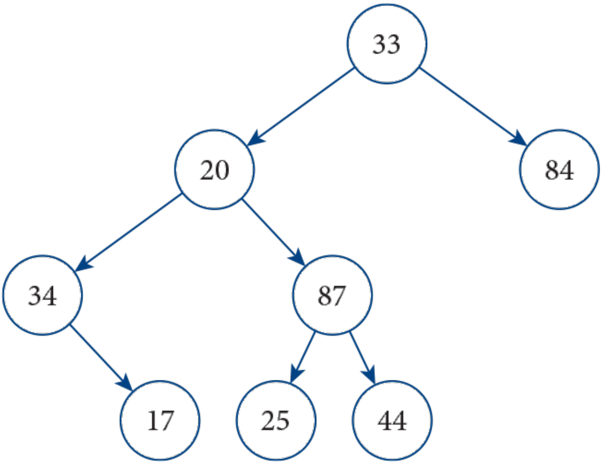
15.10.1 Multiple Choice Exercises

1. root
2. True
3. 0, 1, or 2 children

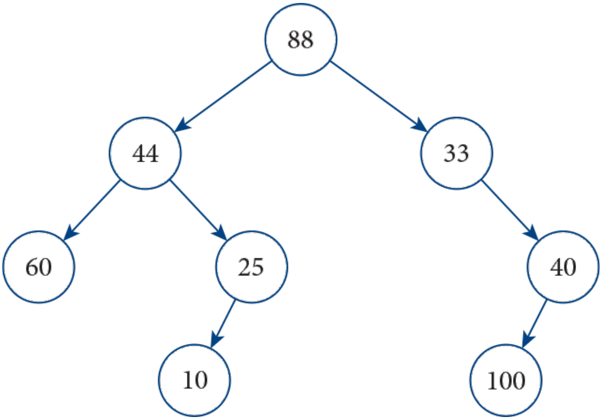
4. *O*(log *n*)

15.10.2 Reading and Understanding Code

1.


33
20
84
34
87
17
25
44

2.


88
44
33
60
25
40
10
100

3. 10, 25, 50, 75

15.10.3 Fill In the Code

1. `nd.getRight( ).setRight( null );`

```
public boolean hasExactlyOne( )
{
  // Your code goes here
  return ( root != null && root.getLeft( ) == null
                        && root.getRight( ) == null );
```

2. `}`

```
public int numberOfOddValues( )
{
  // Your code goes here
  return numberOfOddValues( root );
}
private int numberOfOddValues( IntegerNode nd )
{
  // Your code goes here
  if( nd == null )
       return 0;
  else if( nd.getData( ) % 2 == 1 ) // odd
        return 1 + numberOfOddValues( nd.getLeft( ) )
               + numberOfOddValues( nd.getRight( ) );
  else // not odd
        return numberOfOddValues( nd.getLeft( ) )
               + numberOfOddValues( nd.getRight( ) );
```

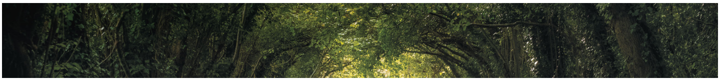
3. `}`

15.10.4 Identifying Errors in Code

1. When the value we are trying to delete is not found, the *private delete* method will eventually return the root of the binary search tree. Unless the tree is empty, the root is not *null* and the *public delete* method will return *true*. Furthermore, the *public delete* method will not properly work either.

15.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

1. We need to test if *nd* is not *null* before calling *getData*, …



Index

Note: Page numbers followed by *f* or *t* indicate material in figures or tables respectively; **bold** page numbers are in Chapters 16, Chapter 17, or Appendix E, which are available in the eBook.

Symbols

&& (ampersands) for AND operator, 164, 164*t*, 165*t*, 235
< > (angle brackets)
  declaring ArrayList objects, 459–460
  defining classes with generic types, 753–758
  greater than operator (>), 163, 163*t*, 165*t*, 194*t*
  greater than or equal to operator (>=), 163, 163*t*, 165*t*, 194*t*
  less than operator (<), 163, 163*t*, 165*t*, 194*t*, 354, 436–437
  less than or equal to operator (<=), 163, 163*t*, 165*t*, 194*t*, 354, 436–437
* (asterisk)
  multiplication operator (*), 37, 50, 50*t*, 52*t*, 63*t*
  shortcut multiplication operator (*=), 61, 62, 63*t*
@ (at sign), for Javadoc block tags, 328, 328*t*
@FXML annotation, **979*t***
@Override annotation, 139, 500, **964**
[ ] (brackets) for declaring arrays, 426–431
, (comma)
  assigning initial values to arrays, 429–431
  in DecimalFormat class pattern, 96*t*
  declaring variables, 38, 429
?: (conditional operator), 192–194, 194*t*
{ } (curly braces)
  assigning initial values to arrays, 349, 429–431
  blocks, 35
  class definition, 278
  method definition, 522
  in selection statements, 168, 169, 171, 178
  try/catch blocks, 566, 567
$ (dollar sign)
  in DecimalFormat class pattern, 96*t*

FXML prefixes, **979*t***
in identifiers, 35
= (equal sign), 63*t*
assignment operator (=), 163, 194*t*, 358
equality operator (==), 162–163, 162*t*, 184–185, 189–190, 194*t*, 363
! (exclamation point)
not equal operator (!=), 194*t*
NOT operator (!), 164, 164*t*, 194*t*, 235
-fx- (FXML prefixes), **979*t***, **1075**
- (hyphen)
decrement operator (– –), 59–60, 62*t*, 63*t*, 194*t*
shortcut subtraction operator (– =), 61, 62, 63*t*, 194*t*
subtraction operator (–), 50, 50*t*, 52*t*, 63*t*, 194*t*
( ) (parentheses), 52, 52*t*, 63*t*, 194*t*, 498
array length, 349
% (percent sign)
in DecimalFormat class pattern, 96*t*
modulus operator (%), 50, 52*t*, 54–56, 63*t*, 194*t*, 259
shortcut modulus operator (%=), 61, 62, 63*t*, 194*t*
. (period), 353
dot notation (.), 81–82
with static class members, 309
in DecimalFormat class pattern, 96*t*
in System class, 108
+ (plus)
addition operator (+), 50, 50*t*, 63*t*, 194*t*
concatenation operator for Strings (+), 86–88
increment operator (++), 59–60, 62*t*, 63*t*, 194*t*
shortcut addition/String concatenation operator (+=), 61, 62, 62*t*, 63*t*, 87, 194*t*
String concatenation operator (+), 194*t*
# (pound sign), in DecimalFormat class pattern, 96*t*
; (semicolon)
in abstract methods, 521, 524
after conditions (error), 169, 170, 245
after statements, 35
after the condition, 216
for loop, 244
/ (slash)
/* */ for block comments, 36
// for line comments, 36
division operator (/), 50, 52*t*, 63*t*, 194*t*
shortcut division operator (/=), 61, 62, 63*t*
_ (underscore), in identifiers, 35

A

abs method (Math class), 107, 110*t*, 111, 180, 186
abstract Axis class, **1071**
abstract classes and methods, 521–529, 543, 597
abstract ShellLinkedList class, 711, 712*t*
Abstract Window Toolkit (AWT), 137
access modifiers, 278–279, 279*t*, 280, 281
  inheritance and, 501, 512–517
  interfaces and, 537
accessor methods, 107, **989**
  writing, 286–289
accumulation, 19, 222–224
  averages, calculating, 227–229, 228*f*
  for loop, 246
ActionEvent class, **975–976**
ActionListener method, **879**, **879*t***
actionPerformed method, **880**
actual parameters, 281
addAll (ObservableList interface), **992***t*
addAll method, **991**, **1073**
addBarChart method, **1069**
addition calculator, 222–224, 223*f*, 238. *See also* accumulation; counting
addition operator (+), 50, 50*t*, 52*t*, 63*t*, 87, 194*t*
  concatenation operator for Strings (+), 86–88
  shortcut addition operator (+=), 61, 62, 62*t*, 63*t*, 87
addListener (ObservableValue interface), **1000*t***
add method
  ArrayList class, 461*t*
  BigDecimal class, 187*t*
  GridPane class, **1005*t***
  Listener method, **875**, **875*t***
  List interface, **1005*t***
add method (Rational class), 312, 313
aggregate array operations
  changing the array size, 361–362
  comparing arrays for equality, 362–363, 363*f*
  copying arrays, 358–361, 360*f*
  displaying data as bar chart, 364–366, 364*f*, 365*f*, 445–448
  maximum/minimum values
    finding, 357–358, 358*f*
    processing columns sequentially, 443–445
  method parameters and return values, 448–451
  multidimensional arrays, 436–448

- printing array elements, 354–355, 355*f*
- processing given column, 440–441
- processing given row, 438–439
- processing rows sequentially, 441–443
- reading data into arrays, 355–356, 356*f*
- searching (*See* searching arrays)
- single-dimensional arrays, 353–366
- sorting (*See* sorting arrays)
- summing array elements, 356–357, 357*f*

Airport class, 323–326
AirportDrawing class, 297, 299
Alert class, **1018**
Algebra class, 313
algorithms, 19
alignment property, **970**
ALU. *See* arithmetic logic unit
American Standard Code for Information Interchange (ASCII), 13
ampersands (&&) for AND operator, 164, 164*t*, 165*t*
anchor point, 150*f*, 151, 153
AND operator (&&), 164, 164*t*, 165*t*, 235
angle brackets < >
- declaring ArrayList objects, 459–460
- defining classes with generic types, 753–758
- greater than operator (>), 163, 163*t*, 165*t*, 194*t*
- greater than or equal to operator (>=), 163, 163*t*, 165*t*, 194*t*
- less than operator (<), 163, 163*t*, 165*t*, 194*t*, 354, 436–437
- less than or equal to operator (<=), 163, 163*t*, 165*t*, 194*t*, 354, 436–437

animate method, 243–244, 595, 616
Animation class, **1024**, ***1025t***, **1029**
AnimationController, **1025–1028**
AnimationSampler application, **1021**, **1024**
anonymous class, **1029**
API. *See* application programming interface
appending to file, 582–584, 582*t*
applets with graphics. *See* graphics
Application class, 497, **963**, ***964t***
application programming interface (API), 77–80
- of Node class, 748*t*

applications, Java, 17
- debugging, 22–24 (*See also* exceptions; testing applications)
- example of (programming activity), 21–24
- executing, 22
- structure of, 34–37

application software, 6
area of a circle, calculation of, 36
argument(s), 18
  methods accepting variable number of, 405–408
  passing arrays as, 448–451
  retrieving command-line arguments, 376–377
  user-defined methods, 281
argument list, 79, 82
arithmetic logic unit (ALU), 3
arithmetic operators, 50–51, 50*t*. *See also* precedence, operator
ArrayIndexOutOfBoundsException exception, 389, 434–435l
ArrayList class, 458–467, 599*t*, 753, **1062**
  declaring and instantiating, 459–460
  example of (programming activity), 468–472
  looping with enhanced for loop, 461–463
  methods of, 460–461
  programming with, 463–467
ArrayList constructors, 459, 459*t*
ArrayListController, 470–471
ArrayList methods, 460–461, 461*t*, 462–463
ArrayPractice2Controller class, 400
ArrayQueue class, 737–738, 735*t*
arrays, 425–476
  applications of, 346
  elements, 346
  index, 346, 349
  length of, 349, 361
  parallel, 378, 380
  queues, representing, 733–739
  recursion, 659–660
  size, changing, 361–362
  stacks, representing, 726–730
arrays, multidimensional, 426–476
  accessing array elements, 431–435
  aggregate array operations, 436–448, 456–457
  declaring and instantiating, 426–431
  displaying data as bar chart, 445–448
  printing array elements, 437–438
  processing columns sequentially, 443–445
  processing given row, 438–439
  processing rows sequentially, 441–443
arrays, single-dimensional, 346–410
  accessing elements of, 349–353, 349*t*
  aggregate array operations, 353–366

  - changing the array size, 361–362
  - comparing arrays for equality, 362–363, 363*f*
  - copying arrays, 358–361, 360*f*
  - as counters, 401–405
  - declaring and instantiating, 346–349
  - displaying data as bar chart, 364–366, 364*f*, 365*f*
  - examples of (programming activity), 367–370, 396–401
  - initial values, 348*t*, 349
  - maximum/minimum values, finding, 357–358, 358*f*
  - printing array elements, 354–355, 355*f*
  - reading data into arrays, 355–356, 356*f*
  - searching arrays
    - Binary Search algorithm, 392–396
    - example of (programming activity), 396–401
    - sequential search, 379, 392
  - searching recursive binary search, 661–664, 797–801
  - sorting elements of, 377–401, 381
    - arrays of objects, 391–392
    - Bubble Sort algorithm, 396–399, 399*f*
    - Insertion Sort algorithm, 381, 385–391, 391*f*
    - Selection Sort algorithm, 381–385, 385*f*
  - summing array elements, 356–357, 357*f*
- ArrayStack class, 727*t*, 728–730
- ASCII. *See* American Standard Code for Information Interchange
- assembly languages, 14–15
- assignment operator (=), 41, 49–50, 52*t*, 194*t*, 358
- assigns, 163
- asterisk (*)
  - multiplication operator (*), 37, 50, 50*t*, 52*t*, 63*t*
  - shortcut multiplication operator (*=), 61, 62, 63*t*
- attachLeft method, 825, 827
- attachRight method, 825, 827
- attributes, **966**
- AudioClip class, **1030**, ***1031t***
- autoboxing, 117, 462–463
- Auto class, 347
  - constructor, 352*t*
  - defining instance variables, 280
  - files generated for, 328*t*
  - method, 352*t*
  - simplified, 329, 330*f*
  - versions of, 283, 287–291, 293–295, 301–302, 304, 307–308
- Auto class API, 468, 468*t*
- autoinitialization, 282

autoReverse, **1028**
averages, calculating, 58–59, 227–229, 228*f*
AWT. *See* Abstract Window Toolkit

B

background-color property, 141, **978**
Backwards class, 257, 257*f*
balanceCheckBook method, 595, 615–616
bang (!)
  not equal operator (!=), 194*t*
  NOT operator (!), 164, 164*t*, 194*t*
bar chart, 136, 136*f*, **1064–1074**
  array data as, 364–366, 364*f*, 365*f*, 445–448
BarChart class, **1064**, **1072*t***
BarChartView class, **1069–1071**
BarChartViewController class, **1073–1074**
base case (recursion), 638. *See also* recursion
  identifying, 638–640
  recursion with two, 652–654
base classes. *See* inheritance; superclasses
beginPath method (GraphicsContext class), 151*t*, 152
BigDecimal class
  comparing floating-point numbers using, 188, 189*f*
  constructor of, 187, 187*t*
  method, 187*t*
Big-Oh, 682, 785–787, 785*t*
Big-Omega, 785, 787
Big-Theta notation, 785, 786
billsBackup array, 358, 358*f*, 359*f*
binary files, 571. *See also* files
binary numbers, 10–12
  hexadecimal numbers for, 12–13, 13*t*
binary operators, 50. *See also* precedence, operator
Binary Search algorithm, 392–396
binarySearch method, 394, 395
binary search of sorted arrays
  recursive approach to, 661–664
  single-dimensional arrays, 392–396
binary search tree
  BinarySearchTreeClient class, 843–844
  code implementation, 836–840
  definition, 829
  deleting, 832–835, 833–835*f*, 842*f*
  example of (programming activity), 846–847

  insert and delete methods, 840, 841
  inserting, 831–832, 832*f*
  linked list, 831, 831*f*
  NullPointerException, 841
  private recursive method, 840
  public wrapper method, 840
  rebalancing, 844–846
  recursive calls, 842*t*
  searching, 830–831, 830*f*
BinarySearchTreeClient class, 843–844
binary trees
  attachLeft method, 825, 827
  attachRight method, 825, 827
  code implementation, 822–825, 827–828
  complete binary tree, 819, 819*f*
  example of (programming activity), 828–829
  full binary tree, 819, 819*f*, 820*t*
  height-balanced, 820, 821*f*
  IntegerNode class, 821–823
  level order traversal, 826
  node class, 818
  in order traversal, 826–827
  post order traversal, 826–827
  pre order traversal, 826–827
  regular tree, 819, 819*f*
  storing integers, 818, 818*f*
  traversal strategies, 825, 826*t*
bits, 4
black box testing method, 181–182
block comments, 36
  including in Javadoc documentation, 328
blocks, 35–36
block scope, 172
block tags, 328
BoardGame class, **1002**–**1004**
BoardView class, **1006**–**1008**
Bonds class, **1068**
Book class, 464, 464–465
BookSearchEngine class, 464, 466
BookStore class, 465–467
boolean data type, 41, 41*t*, 42*t*
boolean expressions, 164
Boolean logic operators. *See* logical operators
boolean variables (flags), 250, 260

boots, 6
BorderedLabelGridView class, **935–936**, **1041–1042**
BorderLayoutView class, **926–927**
BorderPane (JavaFX layout class), **969*t***, **1020–1021**, **1023*f***, **1041**
boundary conditions, 24, 238
boundary values, 181
braces, 35
  assigning initial values to arrays, 429–431
  class definition, 278
  loop body, 215
  method definition, 281, 522, 524
  in selection statements, 168, 169, 171, 178
  try/catch blocks, 566, 567, 570
brackets < >
  declaring ArrayList objects, 459–460
  defining classes with generic types, 753–758
  greater than operator (>), 163, 163*t*, 165*t*, 194*t*
  greater than or equal to operator (>=), 163, 163*t*, 165*t*, 194*t*
  less than operator (<), 163, 163*t*, 165*t*, 194*t*, 354, 436–437
  less than or equal to operator (<=), 163, 163*t*, 165*t*, 194*t*, 354, 436–437
brackets [ ] for declaring arrays, 426–431
break statement. *See* switch statements
Bubble Sort algorithm, 396–399, 399*f*, 801, 813
bull's-eye target, drawing, 253–255, 254*f*, 255*f*
buttons
  components (JavaFX class), **968*t***
  example of (programming activity), **985–988**
byte, 4
byte codes, 15
byte data type, 38, 39, 39*t*, 42*t*, 58*t*

C

C++, 15
calcRating method, 407
calculate method, **974**
calling methods, 18, 76, 79–84, 162
  array elements of objects, 353
  dot notation, 81–82
  example of (programming activity), 93–94
  flow of control, 79, 80*f*
  implicit *vs.* explicit invocation, 502
  processing overhead, 512
  static methods, 106–107

capacity of ArrayList objects, 459
capitalization of identifiers, 22
  classes, 77, 278
  constant, 48
  instance variables, 280
  methods, 77
  variables, 39
carriage return character, 104*t*
Cascading Style Sheets (CSS), **978**, **1075**
case clause. *See* switch statements
case sensitivity, 21–23, 35
cashier application, animation of, 242*f*
CashierApplication class, 264
CashierDrawing class, 242–244, 262–263
catch blocks, 567–575, 592
catching multiple exceptions, 570
CategoryAxis class, **1071**
cellBills array, 350–351, 350*f*, 351*f*, 355–358, 358*f*, 359*f*, 361–362, 373–374
  as bar chart, 364*f*
CellPhone class, 371–373
CellPhoneClient class, 374–375, 376*f*
centimeters, converting inches to, 53–54
central processing unit (CPU), 3–4
  speed of, 4–5
changed (ChangeListener interface), ***999t***
ChangeListener interface, **999**, ***999t***
ChangingColorsController.java, **980**
  application launcher, **981**
changing the array size, 361–362
character class methods, 540*t*
characters. *See also* strings
  ASCII, 13
  Unicode character set, 13, 14*t*
  white space characters, 104, 104*t*
  white space characters, as token delimiters, 587
character wrapper class, 119–120, 119*t*, 120*f*
charAt method (String class), 80, 81*t*, 88–89, 249, 253
char data type, 40, 40*t*, 42*t*, 86
charting array data, 364–366, 364*f*, 365*f*, 445–448
checkboxes, **977–985**
  JavaFX class, ***968t***
checked exception, 567
checkout method (CashierDrawing class), 242, 262–263

ChessBoard class, **910–912**, **1009–1010**
circle class, in Figure hierarchy, 522–524
circle coordinates, conversion, 255, 255*f*
classes, 15, 18, 34, 79, 87–152. *See also* interfaces; *specific class by name*
  abstract, 521–529
  basics and benefits, 76–77
  defining, 278–279, 296–299
  documentation, 328
  encapsulation, 77, 280, 373, 501, 512
  enumerations, 318–323
  example of (programming activity), 121–122
  generic, 753–758
  graphical objects, 316–318
  inheritance and, 496–511
  Java Class Library, 95
  packages of (*See* packages)
  polymorphism, 529–531
  predefined, 95
  reusability, 77
  scope, 280, 285–286, 286*t*
  static members, 106–107, 307–309, 308*t*, 529*t*
  user-defined, 277–330
  wrapper classes, 116–120, 117*t*
.class files, 19
class members, 77, 89, 278, 343. *See also* instance variables; methods; objects
class methods, 106, 306, 308
ClassNotFoundException exception, 611*t*
class scope, 280
class variables, 307
cleaning the data, 606
clear (List interface), ***1005t***
clear method (ArrayList class), 461*t*, **1018**, **1073**
clearRect (GraphicsContext class), 143–144, 143*t*
client programs, 278
clients
  of classes, 76
  in computer networks, 6
clock cycle, 4
close method, 573
  ObjectInputStream class, 610
  PrintWriter class, 582–584
  Scanner class, 587–589

closePath method (GraphicsContext class), 151*t*, 153
closing tag, **966**
code format
  comments (*See* comments)
  indentation of program code, 19
code reusability, 2
colorChosen event handler method, **980**
Color class, 141, 142, 142*t*
Color constants, 141, 142*t*
ColorFrequencyGame class, **932–935**, **1038–1041**
ColorGridGameController class, **937–938**, **1043–1044**
ColorGridGame interface, **931–932**, **1037**
ColorMixer.java, **982**
Color object, 255
colorToHexString method, **977**
column index (multidimensional arrays), 431
columns of two-dimensional arrays
  processing sequentially, 443–445
  processing single column, 440–441
combo boxes, **987–993**
  JavaFX class, ***968t***
comma (,)
  assigning initial values to arrays, 429–431
  in DecimalFormat class pattern, 96*t*
  declaring variables, 429
  defining instance variables, 280
  multiple interfaces, 537
command buttons. *See* JavaFX class
command-line arguments, retrieving, 376–377
comments, 36
  for Javadoc documentation, 328
Comparator interface, 596*t*
compareTo method
  BigDecimal class, 187*t*
  for enum objects, 320*t*, 321
  for sorted linked lists, 739–740
  String class, 190–192, 190*t*, 192*f*, 391
comparing. *See also* conditions
  arrays (for equality), 362–363, 363*f*
  floating-point numbers, 184–189
  objects, 189–192
  strings, 190–192, 190*t*, 192*f*
compiler(s), 15, 20, 22, 47, 48
  errors, 22–23, 24*t*, 48, 170, 178, 247, 281, 285

- javac, 20, 327, 328
- JIT, 16

components, GUI
- BorderPane (JavaFX layout class), **969*t***
- button components (JavaFX class), **968*t***
- checkboxes (JavaFX class), **968*t***
- combo boxes (JavaFX class), **968*t***
- GridPane (JavaFX layout class), **969*t***
- HBox (JavaFX layout class), **969*t***
- labels (JavaFX class), **968*t***
- radio buttons (JavaFX class), **968*t***
- slider (JavaFX class), **968*t***
- StackPane (JavaFX layout class), **969*t***
- text fields (JavaFX class), **968*t***
- VBox (JavaFX layout class), **969*t***

components, nesting, **1035–1046**

composition, 530

compound loop condition, 236–237, 237*f*

computer basics, 3–7

computer networks, 6

computer programming, 2

concatenation operator (+), 45, 63*t*, 194*t*
- shortcut concatenation operator (+=), 61, 62, 62*t*, 63*t*, 87
- for Strings (+), 86–88

conditional operator (?:), 192–194, 194*t*

conditional statements. *See* looping; selection (program flow control)

conditions, 162–168. *See also* comparing; flow of control

console input, 100–106

constants, 46–48. *See also* variables

constructors, 78
- Auto class, 352*t*
- of BigDecimal class, 187*t*
- creating objects using, 78–79
- for DecimalFormat class, 96*t*
- default, 79, 282, 284
- File class, 219, 220*t*
- inheritance rules, 501*t*, 505, 505*t*
- for Random class, 99, 99*t*
- for Scanner class, 101*t*
- of String class, 78*t*, 79
- for subclasses, 502–505
- for user-defined classes, 282–286
- writing, 282–286

Container add method, **922*t***

control flow, 17
controls, GUI, **967**
coordinate system (graphics), 140–142
copying arrays, 358–361, 360*f*
count-controlled loops, 244–257
counters in loops. *See* loop control variables
counting, 224–227, 246
  arrays as counters, 401–405
  averages, calculating, 227–229, 228*f*
  statements, 787–790
  tokens, 249–251, 252*f*
count method, 598, 614
C programming language, 16
CPU. *See* central processing unit
curly braces
  assigning initial values to arrays, 349
  class definition, 278
  loop body, 215
  method definition, 281, 522
  in selection statements, 168, 169, 171, 178
currency, formatting, 115, 115*t*

D

dailyTemp variable, 346
dangling else clauses, 178–180
dash (-)
  decrement operator (– –), 59–60, 62*t*, 63*t*, 194*t*
  shortcut subtraction operator (– =), 61, 62, 63*t*, 194*t*
  subtraction operator (–), 50, 50*t*, 52*t*, 63*t*, 194*t*
data, 34
  hiding, 279
  input (*See* input)
  node, 696
  output (*See* output)
data manipulation methods, writing, 293–296
data representation, 10–14
DataStructureException exception, 712–713
data structures, 696–767
  defined, 696
  linked lists, 696–726, 696*f*
    doubly linked lists, 749–753
    exceptions for, 712–713
    of generic types, 753–758
    implementing queues with, 722–726

    - implementing stacks with, 720–722
    - methods of, 699–705, 713–716
    - nodes, 696–697, 696*f*, 705, 709–710
    - of objects, 707–719
    - of primitive types, 696, 697
    - recursively defined, 758*f*, 759–765
    - sorted, 739–746
    - testing, 705–707, 716–719
  - queues
    - array representation of, 733–739
    - implemented as linked lists, 722–726
  - stacks
    - array representation of, 726–730
    - implemented as linked lists, 720–722
- data types, 37, 42*t*, 45. *See also* variables
  - arrays of (*See* arrays)
  - for assignment, 44*t*
  - boolean, 41, 41*t*
  - character, 40, 40*t*
  - enumeration types, 318–323
  - floating-point, 39–40, 40*t*
  - integer, 39
  - primitive, 38
  - rules of operand promotion, 58*t*
- debugging applications, 22–24, 26–28. *See also* exceptions; testing applications
- DecimalFormat class, 95, 96–99, 229, 304
  - demonstrating, 97–98
  - format method, 96–98, 96*t*
  - special characters for, 96*t*
- decimal representation, converting to binary, 10, 11*t*
- declaring (defining)
  - abstract classes and methods, 521–526
  - ArrayList objects, 459–460
  - arrays
    - multidimensional, 427–428, 456–457
    - single-dimensional, 346–349
  - classes, 278–279, 296–299
  - enumeration types, 318
  - generic classes, 753–758
  - interfaces, 537
  - object references, 78–79
  - subclasses, 500
  - user-defined exceptions, 17

- variables, 38–39, 41–43

decrementing loop control variable, 252
decrement operator (– –), 59–60, 62*t*, 63*t*, 194*t*
default case. *See* switch statements
default constructor, 79, 282, 284
defining. *See* declaring
delimiters, 104
- input, 587

DeMorgan's Laws, 167–168, 167*t*, 235
dequeuing items from queues, 722, 733–737, 735–736*f*, 738
derived class, 496. *See also* inheritance; subclasses
designing programs
- inheritance, 497–511
- iteration *vs.* recursion, 682
- pseudocode, 17–19
- recursion, 638–640, 759–765
  - binary search, 661–664
  - identifying base and general cases, 638–640
  - linked lists defined by, 759–765
- with return values, 641–647
- two base cases, 652–654

deterministic computing, 99
Die class, 402, 403
DieCount class, 403, 404*f*
DieCount2 class, 404–405
direct super- and subclasses, 497, 502, 504
disableButtons method, **1028**, **1029**
divide method, BigDecimal class, 187*t*
division by zero, 56–57
division operator (/), 50, 52*t*, 63*t*, 194*t*
- shortcut division operator (/=), 61, 62, 63*t*

documentation
- class, 328
- with Javadoc, 327–330
- web-style, 329, 330*f*

document object model (DOM), 816
dollar sign ($)
- in DecimalFormat class pattern, 96*t*
- in identifiers, 35

domain name system (DNS), 7
dot/dot notation (.)
- in DecimalFormat class pattern, 96*t*
- in System class, 108

dot notation (.), 81–82, 353

dot operator (.), with static class members, 309
Double class, **1000*t***
double data type, 39–40, 40*t*, 42*t*, 58*t*
  comparing floating-point numbers, 184–189
DoublePredicate interface, 597*t*
double quotes (“ ”), string literals, 44, 45
DoubleStream interface, 596, 601*t*
doubleValue (Double class), **1000*t***
Double wrapper class, 118–119, 118*t*, **975**, **1009**
doubly linked lists, 700*f*, 749–753
do/while loop
  event-controlled looping using, 239–241
  flow of control, 239, 239*f*
  nesting, 260–261
  syntax, 239
  validating input with, 240, 240*f*
DRAM. *See* dynamic random access memory
DrawASprite class, 497
drawGIF method, 579, 580
drawing. *See also* graphics
  custom shapes, 151–153
  methods of GraphicsContext class, 143*t*
  paths, 151–153
  shapes, 146–147, 147*f*
    and displaying text, 148–149, 149*f*
    lines, 144–145, 145*f*
draw method (graphical objects), 316, **1057**, **1059**
drop-down lists. *See* combo boxes
dynamic random access memory (DRAM), 5

E

E constant (Math class), 109–110, 110*t*, 307
element, **966**
  of arrays, 346 (*See also* arrays)
    accessing, 349–353, 349*t*
    summing, 356–357, 357*f*
    swapping, 383
EmailAddress constructor, 617, 619
empty, argument list, 79
encapsulation, 77, 280, 309, **967**
  arrays as instance variables, 373, 374
  inheritance and, 501, 512
endIndex argument, 89
endless loop, 216

enhanced for loops, 461–463
enqueuing items into queues, 723, 723*f*, 724*f*, 733–736, 735–736*f*, 738
enumeration (enum) types, 318–323
EOFException exception, 610–614
equality of arrays, comparing, 362–363, 363*f*
equality operators (==), 162–163, 162*t*, 194*t*, 363
  comparing floating-point numbers, 184–185
  comparing objects, 189–190
equal sign (=), 63*t*
  assignment operator (=), 41, 49–50, 52*t*, 163, 194*t*, 358
  equality operator (==), 162–163, 162*t*, 184–185, 189–190, 194*t*, 363
equalsIgnoreCase method (String class), 190, 190*t*
equals method, 304, 352*t*, 363, 374, 376, 468*t*
  comparing objects, 189–190, 190*f*
  for enum objects, 320*t*, 321
  String class, 190*t*
  user-defined classes, 303–306
error(s). *See also* exceptions; testing applications
  cannot find symbol, 22, 172, 286
  causes of, 24*t*
  compiler, 22–23, 48, 84, 84*f*, 85, 170, 178, 247, 281, 285
  in constructing while loops, 219
  dangling else clauses, 178
  declaring the parameter, 293
  logic, 24, 48, 216, 292
  null object reference, 84, 84*f*, 85
  return type for constructors, 285
  run-time, 23–24, 84
  stack overflow, 646–647
  in writing mutator methods, 292
escape sequences, 45, 46*t*
Euclid's algorithm, 314
even numbers, printing, 248, 248*f*
event-controlled looping, 214–215, 217–222
  text file, reading data from, 219–222
  user, reading data from, 217–219
  using do/while loop, 239–241
  using while loop, 214–215, 217–219
event-driven model, **972**
EventHandler interface, ***1029t***, ***1048t***
event handling, **872–875**, **972**
events, 217, **972**
EventType constants, **1059**
Exception class, 566–567, 568*t*

exceptions, 84, 284, 566. *See also specific exception by name*
  catching multiple (finally blocks), 570
  checked and unchecked, 568
  generating an, 712–71
  handling, 566
  and input/output operations, 565–620
  NullPointerException, 84–85, 283
  recovering from, 577–581
  try/catch blocks, 567–575
  user-defined, 617–619
  when input is not the expected data type, 106*f*
exclamation point (!)
  not equal operator (!=), 194*t*
  NOT operator (!), 164, 164*t*, 194*t*, 235
executing applications, 19
expanding arrays, 361–362
explicit type casting, 58
expressions
  assignment operator and, 49–50
  conditional operator (?:), 192
  switch, 195–196
ExpressShipping class, 547
eXtended Markup Language, 602
extends clause, 497, 537. *See also* inheritance

F

factorials, calculating, 641–647, 784
factoring numbers, 259–261, 261*f*
factory method (Color class), 142, 142*t*
FadeTransition, **1024**, **1028**, **1030**
false (reserved word), 35
familyCellBills Array, 432–435
Fetch-Decode-Execute Cycle, 4
fields, of a class, 76, 278
FIFO. *See* first in, first out
File class, 219, 220*t*, 222, 568, 599*t*
FileInputStream class, 576*t*, 582–584, 610
FileNotFoundException exception, 222, 566, 567*f*, 573, 574, 592, 608, 608*t*, 611*t*, 617
FileOutputStream class, 576*t*, 608*t*
  writing data, 608–610
  writing objects, 608–610
files, 582–592, 607–614. *See also* input; output
  appending to text files, 582–584

  closing, 573
  opening, 573
  pointer, 573
  reading data from text files, 571–575
  reading objects from files, 610–614
  writing data to text files, 582–584
  writing objects to files, 608–610
FileWriter class, 576*t*, 582, 583*t*, 584
fill (GraphicsContext class), 151*t*
fill color, 141
fillGridWithColors method, **934**, **1040**
fillOval (GraphicsContext class), 143*t*, 147–148, 151, 255
fillRect (GraphicsContext class), 143*t*, 146, 151, 365–366, 365*f*, 445–446
fillText (GraphicsContext class), 143*t*, 148
fillWinners method, 379, 381
filter method, 596–598
finally blocks, 566
finding factors, output of, 260–261, 261*f*
findMaximumBill method, 374, 376
fires event, **972**
first in, first out (FIFO), 722
flags (boolean variables), 250, 260
FlatRateShipping class, 547
float data type, 39–40, 40*t*, 42*t*, 58*t*
floating-point data types, 39–40, 40*t*
  comparing floating-point numbers, 184–189
floating point number, **E-1**–**E-3**
floating-point unit (FPU), 3–4, 55
flow of control, 17, 19, 34, 162–201
  calling methods, 79, 80*f*
  looping (iteration)
    do/while loop, 239, 239*f*
    for loops, 245, 245*f*
    while loops, 215, 216*f*
  selection
    conditional operator (?:), 192–194
    if/else if statements, 173–175, 174*f*
    if/else statement, 170–172, 170*f*
    if statements, 168–170, 168*f*
    switch statements, 194–199
  types of, 162
Font class constructor, 149*t*
FoodSamplingsController.java, **988**–**989**

foreground colors, 141
for loops, 392, 407
  ArrayList objects, looping through, 461–463
  basic structure of, 244–245
  constructing, 246–256
  count-controlled loops using, 244–257
  enhanced for loop, 461–463
  flow of control, 245, 245*f*
  multidimensional array operations, 436–448
  nesting, 257–261
  single-dimensional array operations, 354–358, 360–362
  syntax, 244
  testing techniques for, 256–257
formal parameters, 281
format method
  DecimalFormat class, 96–98, 96*t*
  NumberFormat class, 115*t*
format of programming code
  comments (*See* comments)
  indentation, 19
form feed character, 104*t*
forming conditions, 162–168
forward slashes (//), for block comments, 36
FPU. *See* floating-point unit
frame, 643
functional interfaces, 543, 597, 598, **1030**
functions. *See* methods
FXCollections class, **1063*t***
fx:controller (FXML attributes), **973–974**, **979*t***
fx:define (FXML element), **979*t***
fx:id (FXML attributes), **979*t***, **980**
FXML, **962**
  special characters, **995*t***
FXShellApplication, **967*f***

G

garbage collector, 84, 701
gcd method, 647–652
general case (recursion), 638–640, 652–658, 665
generic classes (parameterized types), 458, 753–758
generic Object, 697
generics, 458
  GenericLinkedList class with, 756–758
getAmount method, 616

getBlue (Color class), **998*t***
getChildren (Pane class), **1018*t***
getClass method, **964**
getColor (PixelReader interface), **997*t***
getColumnConstraints (GridPane class), **1005*t***
getCountAutos method, 308–309
getCurrencyInstance method (NumberFormat class), 115*t*
getData (XYChart.series class), **1072*t***
getEventType method, **1059**
getGallonsOfGas method (Auto class API), 352*t*, 468*t*
getGreen (Color class), **998*t***
getGridHexColor method, **1041**, **1045**
getHeight (Image class), **996*t***
getHost method, 619
getImageSelected method, **992**
getIndex method, **1041**, **1045**
getIndexOfLabel method, **938**
getItems (ComboBox class), **991*t***
getLabel method, **935**, **1041**, **1045**
getMessage method (exception classes), 568*t*, 618
get method (ArrayList class), 460–461, 461*t*, 598
getMethodLabel method, **925**
get methods (accessor methods), 287
getMilesDriven method (Auto class API), 352*t*, 468*t*
getModel method (Auto class API), 352*t*, 468*t*
getNext method (CashierDrawing class), 243, 263
getNumber method, 545
getParameters (Application class), **964*t***, **966**
getPercentInstance method (NumberFormat class), 115*t*
getPixelReader (Image class), **996*t***
getPixelWriter (WritableImage class), **997*t***
getPrice method, 263
getReady method, 535
getRed (Color class), **998*t***
getResource method, **964**, **964*t***, **966**
getRowConstraints (GridPane class), **1005*t***
getScene method, **1059**, **1083**
getSceneX (TouchPoint class), **1048*t***
getSceneX method (MouSeevent class), **1047**, **1047*t***
getSceneY (TouchPoint class), **1048*t***
getSceneY method (MouSeevent class), **1047**, **1047*t***
getSelectedIndex (SingleSelectionModel class), **991*t***, **992**
getSelectedItem (SingleSelectionModel class), **991*t***
getSource method (ActionEvent class), **976*t***, **980**, **983**, **1018**, **1030**

getSquareColor method, **1004**
getSquareText method, **1002**
getString method, 603
getStylesheets method, **1076**
getters, 107, 287
getText method, **975**
getTitle method, **926**, **1041**
getTouchPoint method (TouchEvent class), **1047**, ***1048t***
getValue (ObservableValue interface), ***1000t***
getValue (Slider class), **999**, ***999t***
getWidth (Image class), ***996t***
getWindow method, **1059**, **1083**
getX (MouSeevent class), ***1047t***
getX (TouchPoint class), ***1048t***
getY (MouSeevent class), ***1047t***
getY (TouchPoint class), ***1048t***
GIFDecoding class, 577–579
graphical objects, 316–318
graphical user interface (GUI)
  adapter classes, **895**
  building programmatically, **1002**–**1010**
  combo boxes, **892**–**895**
  command buttons, **880**–**883**
  components, **865**–**868**
  container class, **870**, ***870t***
  controls, **967**–**969**
  event handling, **872**–**875**, **972**
  JLabel class, **868**–**871**, ***870t***
  layout containers, **1010**–**1020**
  layout managers
    border layout, **921–929**
    grid layout, **909–921**
  mouse movements, **896**–**909**
  radio buttons and checkboxes, **883**–**889**
  text fields, **875**–**880**
  using GridPane, **1010**–**1020**
  using JavaFX, **961**–**1083**
  using JFrame class, **862**–**865**
  using panels to nest components
    BorderedLabelGridView class, **935–936**
    ColorFrequencyGame class, **932–935**
    ColorGridGameController class, **937–938**
    ColorGridGame interface, **931–932**
    controller class, **930, 931**

        fillGridWithColors method, **934**
        getIndexOfLabel method, **938**
        getLabel method, **935**
        highestCount method, **934**
        PlayColorCount class, **938–939**
        setUpButtons method, **938**
        setUpLabels method, **938**
        UML diagram, **931*f***
        underlying nested layout, **930, 930*f***
graphics, 135–154
    coordinate system and color, 140–142
    drawing shapes, 151–153, 197–199, 199*f*
        bar charts, 364–366, 364*f*, 365*f*, 445–448
        bull's-eye target, 253–255, 254*f*, 255*f*
        ovals (circle), 255
        sprite, 149–150, 150*f*
        and text, 143–151
    example of (programming activity), 154
    JavaFX application structure, 137–140
GraphicsContext
    coordinate system, 140*f*
    methods of, 141*t*
GraphicsContext class, 255, 316, 365–366, 445–446, 579, **1059**
    drawing methods of, 143*t*
        path, 151*t*
gray (Color class), **996**, **997**, **998*t***
greater than operator (>), 163, 163*t*, 165*t*, 194*t*
greater than or equal to operator (>=), 163, 163*t*, 165*t*, 194*t*
greatest common divisor (gcd) calculating, 647–652
GridButton class, **916**–**917**, **1014**–**1015**
GridPane (JavaFX layout class), **969*t***, **1005**, **1010**–**1020**, **1042**
GUIs. *See* graphical user interface

H

handle (EventHandler interface), **1029*t***
handle method, **1008**, **1009**, **1018**, **1030**, **1045**, **1073**
handleMouSeevent method, **1053**
handling events, **972**
handling exceptions, 566–569
    catching multiple (finally blocks), 570
    checked and unchecked, 567
    try/catch blocks, 568–570
    user-defined, 617–619
handwaving method, 792

hard disk, 3, 5*t*
hardware, 3–5
hash (#) , in DecimalFormat class pattern, 96*t*
HashMap class
  code implementation, 475–476
  constructors, 473
  declaring, 473
  instantiating, 473
  java.util package, 473
  methods, 474
hasNextBoolean method (Scanner class), 233*t*
hasNextByte method (Scanner class), 233*t*
hasNextDouble method (Scanner class), 233*t*
hasNextFloat method (Scanner class), 233*t*
hasNextInt method (Scanner class), 232–233, 233*t*, 568
hasNextLong method (Scanner class), 233*t*
hasNext method (Scanner class), 220, 220*t*, 222, 231, 232, 233*t*, 574, 589, 611
hasNextShort method (Scanner class), 233*t*
HBox (JavaFX layout class), **965**, **966**, ***969t***
head of linked list, 697
hexadecimal numbers, 12–13, 13*t*
hexStringColor method, **983**
HexStringHandler class, 545–546
hierarchies, 496
highestCount method, **934**, **1041**
high-level programming languages, 15
hours (Duration class), ***1029t***
HTML tags, 16, 329
hyphen (-)
  decrement operator (– –), 59–60, 62*t*, 63*t*, 194*t*
  shortcut subtraction operator (– =), 61, 62, 63*t*, 194*t*
  subtraction operator, 50, 50*t*, 52*t*, 63*t*
  subtraction operator (-), 194*t*

I

identifiers (names), 35
  classes, 78–79, 278
  generic class, 753
  naming conventions, 77, 278, 280, 289
  reserved words as, 35
  rules for creating, 35*t*
  scope of (*See* scope)
  selection of, 38

if/else if statements, 173–175, 174*f*
if/else statements, 170–172, 170*f*, 402
  conditional operator (?:) *vs.*, 193
  dangling else clauses, 178–180
  example of (programming activity), 182–184
  nested, 177–180
  sequential, 176–177
  testing techniques for, 180–182
if statements, 168–170, 168*f*, 240, 241
IllegalArgumentException exception, 323
IllegalEmailException exception, 617–619
IllegalStateException exception, 573, 575, 579, 589
Image class, **996*t***
ImageView control, **990**
implicit invocation, superclass constructor, 502
implicit parameter, 300
implicit type casting, 57
import statement, 95
inches, converting to centimeters, 53–54
incorrect loop condition, 235
incorrect statement, 169
incrementing loop control variable, 248–249
increment operator (++), 59–60, 62*t*, 63*t*, 194*t*
indentation of program code, 19
  if/else statement, 171
  if statement, 169
index, array, 346, 349, 431
indexes, 82–83
indexOfComma, 91–92
indexOfLargestElement method, 384
indexOf method (String class), 80, 81*t*, 83, 89, 91
indexOfWinner method, 379–380, 392
induction, proof by, 793
inequality. *See* not equal operator (!=); relational operators
infinite loop, 216
infinity value, 56
inheritance, 496–511, **967**
  abstract classes and methods, 521–529
  designing, 497–511
  example of (programming activity), 521–529
  exceptions, 566, 576
  inherited members of a class, 498–502
  interfaces, 537–549
  overriding methods, 508–511

  - polymorphism, 529–531
  - protected access modifier, 512–517
  - subclasses, 497
    - constructors, 502–505
    - specialization, 506–508
  - superclasses, 497
  - syntax (extends), 497
- init (Application class), **964*t***
- initialize method, **990**, **992**, **997**, **1024**, **1028**
- initial values, 41–44
  - arrays, 348*t*, 349, 428–431
  - instance variables, 280, 282, 282*t*
- inline tags, 328
- inner class, **999**
- in order and post order traversal methods, 827
- input
  - class, 576*t*
  - reading from Java console, 100–106
  - reading from text files, 571–575
  - reading objects from files, 610–614
  - retrieving command-line arguments, 376–377
  - type-safe input, 232–234
- InputMismatchException exception, 105, 232, 233, 567*f*, 589
- InputStream class, 576*t*, 582
- InputStream dataSource, 101*t*
- insertion sort, 381, 385–391, 391*f*, 800–801
- instance
  - of class, 76 (*See also* objects)
  - method, 308*t*, 309
- instanceof operator, 304, 305
- instance variables, 76, 292
  - access modifiers, 280, 501, 512–517
  - arrays as, 373, 374
  - defining, 279–280
  - inherited, 498–502, 501*t*, 512–517
  - initial values for, 280, 282, 282*t*
  - protected (access modifier), 512–517
  - selection criteria, 280
  - syntax, 279
  - transient, 610
- instantiation
  - of arrays, 428–429
    - single-dimensional, 346–349
  - of objects, 15, 34, 76, 78–79, 80*f*

  using constructors, 78–79
Instruction Register, 4
instructions (programming steps), 2, 4, 17, 18, 34
int data type, 39, 39*t*, 42*t*, 58*t*
integer data types, 39
integer division, 54–56
IntegerLinkedList class, 699*t*, 702–703
IntegerLinkedListTest class, 705–706
IntegerNode class, 697, 821–823
integer, reading, 232
integer unit (IU), 3
Integer wrapper class, 118, 118*t*
Integrated Development Environments (IDEs), 21–23, 25, 37
interfaces, 537–549
Internet, 3, 6–7
IntStream interface, 596
invalid, 49
invertDelta method, **925**
InvestmentApplication class, **1074**
Investment class, **1066–1067**
IOException exception, 222, 566, 567, 608*t*, 644*t*, **965**, **966**
IP address, 6
"is a" relationships, 497
isBlank method (String class), 81*t*, 89
isCorrect method, **1041**, **1045**
isEmpty method (ShellLinkedList class), 712
ItemEvent class, ***885t***, **889**
itemSelected method, **989**
iteration, 19, 215, 792–793
IU. *See* integer unit

J

JAR file, making, 24–25
Java
  applets, 16
  application structure, 34–37
  escape sequences, 45, 46*t*
  keywords, A-1
  language, 16–17
  letters, 35
  words, A-1
Java application, developing, 19–21
Java ARchive (JARs), 24–25
javac (Java compiler), 20, 327

Java Class Library, 95, 101
Java console, reading from, 100–106
Java Development Kit (JDK), 20, 327
Javadoc, 327–330
.java files, 19
JavaFX class
  application structure, 137–140, **962**–**967**
  coordinate system and color, 140–142
  drawing custom shapes, 151–153
  drawing shapes and text, 143–151
  graphical user interfaces (GUIs) using, **961**–**1083**
  hierarchy of controls, **968*f***
  style sheet for SimpleMathPractice class, **1075**
JavaFX layout class
  BorderPane class, **969*t***, **1041**
  GUI components, **965**, **966**, **969*t***, **1043**
  StackPane class, **969*t***
  VBox class, **969*t***, **970**, **1024**
javafx.scene.paint package, 141
Java identifiers, 35, 35*t*
java.io package, 95*t*, 566, 575–577, 582–584, 608
java.lang package, 86, 95*t*, 107, 109, 319, 566, 613
java.math package, 187
java.nio.file package, 598
Java packages, 95, 95*t*
  user-defined, 327
JavaScript Object Notation (JSON), 602
Java SDK, 137
Java servlets, 16
java.text package, 95*t*, 96, 114
java.util package, 95*t*, 99, 101, 458, 575, **1028**, **1029*t***
Java Virtual Machine (JVM), 15, 16, 281, 307
  exception handling, 56, 566, 567, 570
  executing Java code, 19–20, 162
  invoking, 22
  memory allocation, 79
  method calls and recursion, 646
  polymorphism, 529–531
  transfers control to the method, 80
JButton class, **880**–**882**
JCheckBoxes class, **884*t***, **885*t***, **887**–**889**
JComboBox class, **892**–**894**, **892*t***
JComponent class, **930**
JDK. *See* Java Development Kit

JFrame class, **862–865**, **901**, **901*f***
jGRASP, 37
JIGraphicsUtility class, 137–139
JLabel class, **868–871**, **870*t***
JPanel class, **930**
JRadioButtons class
  color selection, **885–886**
  constructors, **884*t***, **885*t***, **886**
  example of (programming activity), **890–892**
jshell utility, 37
JSONArray class, 604*t*
JSONObject class, 604*t*
JSON string, parsing, 604*t*
just-in-time (JIT) compilers, 16
JVM. *See* Java Virtual Machine

K

keywords, Java, 22, 35

L

labels (JavaFX class), **968*t***
lambda expressions, 543, 598, 600, 614, **1030**, **1048**
last in, first out (LIFO), 720
launch (Application class), 139, **964*t***
layout containers, **1010–1020**
  example of (programming activity), **1077–1080**
layout managers
  border layout
    addToRed method, **925**
    BorderLayoutView class, **926–927**
    constructors, **922*t***
    Container add method, **922*t***
    getMethodLabel method, **925**
    getTitle method, **926**
    invertDelta method, **925**
    MixColors class, **929**
    RGBColorMixer class, **923–925**
    RGBColorMixerController class, **927–928**
    UML diagram, **923*f***
  example of (programming activity), **940–943**
  flow layout, **909**
  grid layout
    actionPerformed method, **912**
    ChessBoard class, **910–912**

        - constructors, **910*t***
        - GridButton class, **916–917**
        - PlayTilePuzzle class, **920–921**
        - possibleToPlay method, **916**
        - setUpGame method, **916**
        - TilePuzzle class, **912–915, 918*f***
        - TilePuzzleViewController Class, **918–920, 918*f***
        - won method, **916**
- L1 cache, 4, 5*t*
- L2 cache, 4–5, 5*t*
- length method (String class), 81*t*, 82–83, 88
- length of arrays, 349, 361, 431–432
- less than operator (<), 163, 163*t*, 165*t*, 194*t*
- less than or equal to operator (<=), 163, 163*t*, 165*t*, 194*t*
- LIFO. *See* last in, first out
- line feed character, 104*t*
- lines method, 598
- lineTo method (GraphicsContext class), 151*t*, 152
- linked lists, 696–726, 696*f*
    - defined, 696
    - doubly linked lists, 749–753
    - generic types, 753–758
    - implemented queues with, 722–726
    - implementing stacks with, 720–722
    - methods of, 699–705, 713–716
    - nodes, 696–697
    - of objects, 707–719
    - recursively defined, 758*f*, 759–765
    - sorted, 739–746
        - example of (programming activity), 746–749
    - testing, 705–707, 716–719
- LinkedListTest class, 757–758
- listener, **972**
- literal values, 41
- load method (FXMLLoader class), **965*t***
- local area network (LAN), 6
- local scope, 285–286
- logical operators, 164–167, 164*t*
    - DeMorgan’s Laws, 167–168, 167*t*, 235
    - loop conditions, constructing, 235
    - truth table for, 164, 164*t*
    - working of, 165–166
- logic errors, 24, 24*t*, 48, 216, 566
    - No-op effect, 292

login procedure, 86
log method (Maths class), 110*t*
long data type, 39, 39*t*, 42*t*, 58*t*
LongStream interface, 596
loop body, 215
loop continuation condition, 234, 235
loop control variables, 246, 248–249, 249*f*, 252, 259
  decrementing, 252
  incrementing, 248–249
  scope of, 247
looping, 19, 162, 213–264, 214
  array operations
    multidimensional arrays, 436–448
    single-dimensional arrays, 354–358, 360–362
  conditions for
    compound, 236–237, 237*f*
    constructing, 234–237
    incorrect, 235
    loop continuation condition, 234, 235
    loop termination condition, 234, 235
    steps in, 235
  count-controlled loops using for loop, 244–247
  do/while loop, 239–241
  endless loop, 216
  event-controlled, 214–215, 217–222
    text file, reading data from, 219–222
    user, reading data from, 217–219
    using do/while loop, 239–241
    using while loop, 214–215, 217–219
  for loops, 392, 407
    basic structure of, 244–245
    constructing, 246–256
    count-controlled loops using, 244–257
    example of (programming activity), 262–264
    flow of control, 245, 245*f*
    nesting, 257–261
    single-dimensional array operations, 354–358, 360–362
    syntax, 244
    testing techniques for, 256–257
    two-dimensional array operations, 436–448
  infinite loop, 216
  iteration, 215
  loop update statement, 216
  nested loops, 257–261

  - recursion *vs.*, 682
  - scanner, type-safe input using, 232–234, 234*f*
  - techniques for, 222–231
    - accumulation, 222–224
    - average, 227–229, 228*f*
    - counting, 224–227, 403–405
    - finding maximum/minimum values, 229–231
  - through ArrayList objects, 461–463
  - while loops, 257, 394
    - example of (programming activity), 241–244
    - nesting, 257–261
    - testing techniques for, 237–238
  - zero iterations, 215
- loop termination condition, 234, 235
- loop update statements, 216
  - for loops, 244–247, 249, 253

M

- machine language, 14
- Mac OS, system information, 7, 9*f*
- mapToDouble method, 600, 614
- Math class, 95, 106–107, 109–114
  - abs method, 107, 111, 180, 186
  - methods of, 110, 110*t*
  - min and max methods, 113–114, 113*f*
  - pow method, 111–112, 112*f*
  - round method, 112–113, 113*f*
  - static constants of, 107, 110*t*, 119
- maximum values, finding, 176–177, 229–231
  - among array elements, 357–358, 358*f*
- max method (Math class), 110*t*, 113–114, 113*f*
- members of classes, 77, 278. *See also* instance variables; methods; objects
  - inherited, 498–502
  - interface members, 537
  - protected (access modifier), 512–517
  - static, 307–309, 308*t*
- memory, 3–5, 4*t*, 5*t*
  - allocation, 347
- merge sort, 670–676
- meta-characters, 587
- methods, 15, 18, 35, 76
  - abstract, 521–529
  - arrays as parameters, 448–451

- of BigDecimal class, 187*t*
- calling, 18, 76, 79–84, 80*f*, 162
  - constructors, 502–505
  - example of (programming activity), 93–94
  - implicit vs. explicit invocation, 502
- of Character wrapper class, 119*t*
- of Double wrapper class, 118*t*
- for enum objects, 320, 320*t*
- of GraphicsContext Class, 141*t*
- inherited, 498–502
- of Integer wrapper class, 118*t*
- of Math class, 110*t*
- non-static, 308*t*, 309
- of NumberFormat class, 115*t*
- overloading, 284, 510, 511, 511*t* (*See also* inheritance)
  - overriding vs., 510, 511, 511*t*
- overriding, 303, 498, 508–511
  - overloading vs., 510, 511, 511*t*
- passing an argument to, 18
- of PrintStream class, 108*t*
- protected (access modifier), 500
- public *vs.* private, 81
- Scanner class, 219, 220*t*
- of Scanner class, 101*t*
- static, 308, 308*t*
- user-defined, 280–282
  - accessor methods, 286–289
  - constructors, 282–286
  - data manipulation methods, 293–296
  - exceptions, 617–619
  - mutator methods, 289–293
  - *this* reference, 299–303
  - toString and equals methods, 303–306

milesPerGallon method (Auto class API), 295, 352*t*, 468*t*

millis (Duration class), **1029*t***

minimum values, finding, 176–177, 177*f*, 229–231
- among array elements, 357–358

min method (Math class), 110*t*, 113–114, 113*f*

minus (–)
- decrement operator (– –), 59–60, 62*t*, 63*t*
- shortcut subtraction operator (– =), 61, 62, 63*t*
- subtraction operator (–), 50, 50*t*, 52*t*, 63*t*

minutes (Duration class), **1029*t***

MixColors class, **929**

mixed-type arithmetic, 57–59
MixingColorsController controller, **983**–**984**
mix method, **983**
Model-View-Controller architecture, **972**
modulus operator (%), 50, 52*t*, 54–56, 63*t*, 259
  shortcut modulus operator (%=), 61, 62, 63*t*
moneySpentOnGas method, 295
monthly temperatures, graphical representation of, 136–137, 136*f*
MouSeevent class, **896**, **896*t***, **902**, **1047**
mouse events, **1047**–**1061**
MouseListener interface, **896**, **896*t***
MouseMotionListener interface, **903**, **904*t***
mouse movements
  getInsets method, **901**, **902*t***
  Insets class, **901**, **902*t***
  model-view-controller architecture, **897**
  MouseAdapter class, **896**
  mouseClicked method, **901**
  MouSeevent class, **896**, **896*t***, **902**
  MouseListener interface, **896**, **896*t***
  MouseMotionListener interface, **903**, **904*t***
  mouseMoved method, **908**
  PlaySubHunt class, **903**
  repaint method, **902**, **902*t***
  setTitle method, **902**
  SubHunt class, **897**–**899**
  SubHunt draw method, **902**
  SubHuntViewController class, **899**–**901**
  translate method, **902**, **903*t***
moveTo method (GraphicsContext class), 151*t*, 152
Movie Record class, 605
MovieWinners class, 378–380, 392
multidimensional arrays, 425–476
  accessing elements of, 431–435
  aggregate operations, 436–448
    displaying data as bar chart, 445–448
    printing array elements, 437–438
    processing all elements, 436–438
    processing columns sequentially, 443–445
    processing given column, 440–441
    processing given row, 438–439
    processing rows sequentially, 441–443
  assigning initial values, 429–431
  declaring and instantiating, 426–431

  example of (programming activity), 451–456
multiplication operator (*), 37, 50, 50*t*, 52*t*, 63*t*
  shortcut multiplication operator (*=), 61, 62, 63*t*
multiply method (Rational class), 312
mutator methods, 107, **980**
  public, 289
  writing, 289–293

N

name collisions, **967**
name precedence, 292
namespace, **967**
NaN. *See* Not a Number
n-dimensional array, structure of, 456, 456*t*
negative integer, D-1–D-2
nested *if/else* statements, 177–180
nested loops, 257–261
nesting
  loops, 257–261
  processing multidimensional arrays, 436–448
networks, 3
new keyword
  instantiating ArrayList objects, 459–460
  instantiating arrays, 347, 348
newRatings parameter, 407
nextBoolean method (Scanner class), 101*t*, 103, 220*t*
nextByte method (Scanner class), 101*t*, 103, 220*t*
nextDouble method (Scanner class), 101*t*, 103, 220*t*
nextFloat method (Scanner class), 101*t*, 103, 220*t*
nextInt method (Random class), 99–100, 99*t*
nextInt method (Scanner class), 101*t*, 103, 104, 220*t*, 222, 568, 574
nextLine method (Scanner class), 101*t*, 105, 233, 233*t*, 251, 574
nextLong method (Scanner class), 101*t*, 103, 220*t*
next method (Scanner class), 101*t*, 103, 220*t*
nextShort method (Scanner class), 101*t*, 103, 220*t*
Node class, **1048**, **1059**
  using generics, 754–755
non-static methods, 308*t*, 309
No-op effect, 292
NoSuchElementException exception, 573, 575, 579, 589
Not a Number (NaN), 56
not equal operator (!=), 194*t*
NOT operator (!), 164, 164*t*, 194*t*, 235
null object references, 84–85, 85*f*, 282, 299, 348

NullPointerException exception, 84–85, 283, 353, 353*f*, 435, 567, 567*f*, 592, 704, 722, 726, **966**
NumberAxis class, **1071**
Number class, **1069**
NumberFormat class, 95, 114–116, 115*t*, 116*f*, 357
NumberFormatException exception, 567*f*, 570, **975**

O

object, 34
Object class, 189, 303, 363, 461, 497–498, 500, 537, 757
ObjectInputStream class, 576*t*, 610
object-oriented programming (OOP), 15–16, 76
ObjectOutputStream class, 576*t*, 608–610, 608*t*
object references, 76, 78–79, 80*f*, 82, 84–85
  null object references, 84–85, 85*f* (*See also* null object references)
  this reference, 299–303
objects, 15
  array of, 347, 351–352
  arrays of, 426–431
    sorting, 391–392
  comparing
    equals method, 189–190, 190*f*
    Strings method, 190–192, 190*t*, 192*f*
  creating
    with constructors, 78–79
    with factory methods, 115
  enum, 319–320, 320*t*
  generic, 697
  instantiation, 76, 78–79, 80*f*
objects (*continued*)
  reading and writing to files, 607–614
    example of (programming activity), 614–617
  this reference, 299–303
object wrappers, 95
observableList (FXCollections class), **1062**
ObservableList method, **1062**, **1064**, **1083**
ObservableValue interface, **1000*t***
offsets, 151
onAction (FXML attributes), **979*t***, **1029**, **1030**
OOP. *See* object-oriented programming
opening file, 573
operands, 49
  promotion, 57, 58*t*
operating system (OS), 5–6

  configuration, 7–10
operator precedence, B-1–B-2
operators
  arithmetic operators, 50–51, 50*t*
  assignment operator (=), 41, 49–50
  binary, 50
  conditional operator (?:), 192–194
  equality operators, 162–163, 162*t*
  instanceof, 304, 305*t*
  logical operators, 164–167, 164*t*
  precedence, 51–53, 52*t*, 63*t*
  precedence of, 164*t*–165*t*, 194, 194*t*
  relational operators, 163, 163*t*
  shortcut, 59–65, 62*t*–63*t*
order of magnitude, 682, 785–787, 785*t*
order of operator precedence, 52, 52*t*, 63*t*
ordinal method (enum objects), 320*t*, 321
org.json package, 603
OR operator (||), 164, 164*t*, 165*t*, 235
OS. *See* operating systems
output
  class, 576*t*
  writing objects to files, 608–610
  writing to text files, 608–610
    appending to text files, 582–584
OutputStream class, 576*t*, 608
overloading methods, 284, 510, 511, 511*t*. *See also* inheritance
  overriding *vs.*, 510, 510*t*, 511, 511*t*
overriding methods, 303, 498, 508–511
  inheritance and, 508–511
  overloading *vs.*, 510, 510*t*, 511, 511*t*

P

packages, 95, 95*t*
  user-defined, 327
palindrome search (example), 654–658
Pane class, **1018*t***
parallel arrays, 378, 380
ParallelTransition, **1024**, **1032–1033**, **1034*t***
parameterized types, 458, 753–758
Parent, **967**, **1035**
parentheses ( ), 52, 52*t*, 63*t*, 81, 82, 194*t*, 498
  array length, 349
  calling methods, 81

parse, 587–589
parseDouble method (Double class), 118*t*, **975**
parseInt method (Integer wrapper class), 118*t*
parsing a String using Scanner, 587–589
passing an argument to method, 18. *See* argument(s)
passing test scores, counting, 225–226, 226*f*
PasswordSecurityHandler class, 540
PasswordSecurityHandlerClient class, 542
Paths class, 598, 599*t*
paths, drawing, 151–153
percent sign (%)
  in DecimalFormat class pattern, 96*t*
  shortcut modulus operator (%=), 61, 62, 63*t*
performance
  of CPU, 4
  iteration *vs.* recursion, 682
  protected access modifier, 512
period (.), 353
  in DecimalFormat class pattern, 96*t*
  dot notation, 81–82
  with static class members, 309
  in System class, 108
Perl, 15
personal computer (PC), design of, 3
PHP, 15
PI constant (Math class), 109–110, 110*t*, 307
PieChart class, **1062**–**1063**, **1064*f***
pixel, 140–141
PixelReader interface, **996*t***
PixelWriter interface, **997*t***
play (AudioClip class), **1031*t***
PlayColorCount class, **938**–**939**, **1045**–**1046**
Player class, 697, 707–709
PlayerLinkedList class, 714–715, 714*t*
PlayerLinkedListTest class, 716–719
PlayerNode class, 709–710
  for doubly linked list, 750
PlayerQueueLinkedList class, 723*t*, 724–726
PlayerSortedLinkedList class, 741, 741*t*, 742–744
PlayerSortedLinkedListTest class, 745
PlayerStackLinkedList class, 720*t*, 721–722
play head, **1034**
play method, **1024**, **1053**, **1057**, **1059**
PlaySubHunt class, **1053**

PlayTilePuzzle class, **920–921**, **1019**
PlayTreasureHunt class, **1060–1061**
plus (+)
  addition operator (+), 50, 50*t*, 52*t*, 63*t*
  concatenation operator for Strings (+), 86–88
  increment operator (++), 59–60, 62*t*, 63*t*
  shortcut addition operator (+=), 61, 62, 62*t*, 63*t*, 87
polymorphism, 529–531
  example of (programming activity), 531–536
popping items from stacks, 720–722, 726–730
possibleToPlay method, **916**, **1014**
postfix operators, 60–61
pound sign. *See* hash (#)
power of 2, 11–12, 11*t*
pow method (Math class), 110*t*, 111–112, 112*f*
precedence
  name, 292
  operator, 51–53, 52*t*, 63*t*
  order of, 52
predefined classes, 95
  example of (programming activity), 121–122
Predicate interface, 597, 597*t*
prefix operators, 60–61
priming read, 217, 219, 222
primitive data types, 38, 278, 391
  wrapper classes for, 116, 117*t*
printHelloWorldNTimes method, 639
printing array elements, 354–355, 355*f*
println method (PrintStream class), 108–109, 108*t*
println method (PrintWriter class), 582–584
print method (PrintStream class), 108–109, 108*t*
printStackTrace method (exception classes), 568
PrintStream class, 95, 108–109, 108*t*
PrintWriter class, 576*t*, 582–584, 583*t*
private (access modifier), 279, 279*t*, 280
  inheritance and, 500
  UML diagrams (indicating on), 498
private recursive method, 840
processDigit method, 539
processing, String, 90–93
processLetter method, 539
processors, 3
processOther method, 539
Program Counter, 4

program design
  inheritance, 497–511
  pseudocode, 17–19
  recursion, 638
  UML diagrams, 496, 498
programming basics, 17
programming languages
  assembly languages, 14–15
  high-level languages, 15
  machine language, 14
  object-oriented programming language, 15–16
promotion of operands, 57, 58*t*
prompts, 102–103, 103*f*
proof by induction, 793
protected (access modifier), 279, 279*t*, 512–517
  rules for, 516–517*t*
pseudocode, 17–19
pseudorandom numbers, 99
public (access modifier), 279, 279*t*, 282
  inheritance and, 500, 501*t*
  interfaces, 537
  UML diagrams (indicating on), 498
public mutator method, 289
public wrapper method, 840
pushing items into stacks, 720–722, 726–730
Python, 15

Q

question mark for conditional operator (?:), 192–194
queues
  array representation of, 733–739
  defined, 722, 722*f*
  implemented with linked lists, 722–726
quick sort, 676–681

R

radio buttons, **977–985**
  JavaFX class, **968*t***
radio buttons vs.checkboxes, **884*t***
Random class, 95, 99–100
random numbers, generation of, 99–100
Rational class, 309–315
rational number, representation of, 309
reading data into arrays, 355–356, 356*f*

readObject method (ObjectInputStreamclass), 610–614
record class, 590
recursion, 638–682
  array, 659–660
  binary search, 661–664
  identifying base and general cases, 638–640
  iteration *vs.*, **990**–**991**
  linked lists defined by, 758*f*, 759–765
  merge sort, 670–676
  palindrome search (example), 654–658
  quick sort, 676–681
  with return values, 641–647
    calculating factorials, 641–647
    calculating greatest common divisor, 647–652
  Towers of Hanoi (example), 665–670
  two base cases, 652–654
recursiveBinarySearch method, 662
RecursiveLinkedList class, 759*t*, 760–761
RecursiveLinkedListTest class, 763–764
recursive method, 638
recursivePalindrome method, 656
recursiveTOfH method, 667, 669
referenced, 172
  access modifiers, 279
Region, **967**
register, **974**
regular expression, 587
relational operators, 163, 163*t*
remote server, reading formatted open data from
  accessing remote data, 601–602
  JSON formatting and parsing, 602–605
  reading, parsing, streaming, and processing remote data, 605–607
removeListener (ObservableValue interface), **1000*t***
remove method (ArrayList class), 461*t*
repetition, 214
reserved words, 35
resources, 573
return statement, 281
  in recursion methods, 641
return values (methods), 18, 80
  arrays as, 374, 448–451
  for constructors (error), 285
  recursion with, 641–647
reusability of code, 77

  generic classes, 458
  through interfaces, 537
RGB color, 141, 142*t*
RGBColorMixer class, **923–925**
RGBColorMixerController class, **927–928**
rollCount array, 403, 405
RotateTransition, **1024**, **1030–1031**, **1032*t***
rounding errors, 184
round method (Math class), 110*t*, 112–113, 113*f*
routers, 6
row index (multidimensional arrays), 431
rows of multidimensional arrays
  processing rows sequentially, 441–443
  processing single row, 438–439
rules of scope, 286, 286*t*
running time analysis, 783–807, 785*t*
  counting statements, 787–790
  evaluating recursive methods, 791–795
  of searching and sorting algorithms, 797–801
  sorting algorithms, 802–807
run-time errors, 23–24, 24*t*
runTransition method, **1029**

S

ScaleTransition, **1024**, **1032**, **1033*t***, **1082**
Scanner class, 95, 251, 568, 571, 571–572*t*, 576*t*, 587*t*, 602*t*
  for character input, 103–104
  constructor, 101*t*
  hasNext method, 574
  methods for testing tokens, 233*t*
  parsing a String, 587–589
  reading data from text files, 219–222, 587–592
  reading user input from the Java console, 100–106
  type-safe input using, 232–234, 234*f*
Scene class, **965**, **965*t***, **1076**, **1083**
scene graph, **962**
scope, 172
  block scope, 172
  class scope, 280
  local scope, 285–286
  loop control variable, 247
  rules of, 286, 286*t*
searching arrays
  Binary Search algorithm, 392–396

  example of (programming activity), 396–401
  recursive binary search, 661–664, 797–801
  sequential search, 378–381, 392
seconds (Duration class), **1029*t***
securityLevel method, 542
Seed value, 99
select (SingleSelectionModel class), **991*t***
selection (program flow control), 18, 19, 162
  conditional operator (?:), 192–194
    vs. if/else, 193
  dangling else clauses, 178–180
  if/else if statements, 173–175, 174*f*
  if/else statements, 170–172, 170*f*
    example of (programming activity), 182–184
    nested, 177–180
    sequential, 176–177
  if statement, 168–170, 168*f*
  testing techniques, 180–182
Selection Sort (arrays), 381–385, 385*f*, 797
selectionSort method, 384
Seller class, 406–407
SellerRatings class, 407–408, 408*f*
semicolon (;)
  abstract method definition, 537
  in abstract methods, 521
  after conditions (error), 169, 170, 245
  after the condition, 216
  for loop, 244
sentence backward, printing, 252–253, 253*f*
sentinel value, 215, 217–219, 218*f*, 222–224, 226, 228–229, 234, 238
sequential execution, 162
sequential *if/else* statements, 176–177
sequential processing, 17
sequential search, 379
  of sorted arrays, 392
  of unsorted array, 378–381
Sequential Search algorithm, 790
sequentialSearch method, 400
SequentialTransition, **1024**, **1032**, **1033*t***
Serializable interface, 608, 609, 611*t*
servers, computer networks, 6
servlets, 16
setAutoReverse (Animation subclasses), **1025*t***
setButtonListener method, **1073**

setByAngle (RotateTransition class), **1032*t***
setColor (PixelWriter interface), **997*t***
setContentText (Alert class), **1019*t***
setCycleCount (Animation subclasses), **1025*t***
setCycleCount (AudioClip class), **1031*t***
setDenom method (Rational class), 312
setFill method (GraphicsContext class), 141, 141*t*
setFont (GraphicsContext class), 143*t*
setFromAngle (RotateTransition class), **1032*t***
setFromValue (FadeTransition class), **1028*t***
setFromX (TranslateTransition class), **1033*t***
setFromY (TranslateTransition class), **1033*t***
setGallonsOfGas method (Auto class API), 300–302, 352*t*, 468*t*
setHeaderText (Alert class), **1019*t***
setLabelBackground method, **1045**
setLineWidth (GraphicsContext class), 143*t*, 146
setMaxHeight (Button class), **1008*t***
setMaxWidth (Button class), **1008*t***
set method (ArrayList class), 461*t*
set methods (mutator methods), 107, 289
setMilesDriven method (Auto class API), 352*t*, 468*t*
setModel method (Auto class API), 352*t*, 468*t*
setName (XYChart.series class), **1072*t***
setOnAction (Button class), **1008*t***
setOnFinished (Animation subclasses), **1024**, **1025*t***
setOnMouseMoved (Node class), **1049*t***
setOnMousePressed (Node class), **1049*t***
setOnTouchMoved (Node class), **1049*t***
setOnTouchPressed (Node class), **1049*t***, **1053**
setPercentHeight (RowConstraints class), **1006*t***
setPercentWidth (ColumnConstraints class), **1006*t***
setScene (Stage class), **965*t***
setStroke method (GraphicsContext class), 141, 141*t*
setStyle method (Button class), **1008*t***, **1009**
setStyle method (Label class), **980**
setters, 107, 289
setText method, **975**
setTitle (Alert class), **1019*t***
setTitle (Stage class), **965*t***, **1051–1052**, **1059**
setToAngle (RotateTransition class), **1032*t***
setToValue (FadeTransition class), **1028*t***
setToX (ScaleTransition class), **1032*t***
setToX (TranslateTransition class), **1033*t***
setToY (ScaleTransition class), **1032*t***

setToY (TranslateTransition class), **1033*t***
setUpButtons method, **938**, **1045**
setUpGameGUI method, **1017**, **1018**
setUpGame method, **916**, **1013**, **1045**
setUpGraphics method, 137, 138*t*, 139, 146
setUpLabels method, **938**, **1045**
setVgrow method, **1043**
ShellApplication, 34–35, 37
ShellGraphicsApplication class, 137–140
ShellLinkedList class, 710–712, 712*t*
  using generics, 755–756
ShippingCalculator class, 548–549
ShippingCost interface, 546
short-circuit evaluation, 166, 704
shortcut operators, 59–65, 62*t*–63*t*
short data type, 39, 39*t*, 42*t*, 58*t*
show (Stage class), **965*t***
showAndWait (Alert class), **1019*t***
showMessageDialog method, **1019**
signature, method, 284
  overloading, 284
  overriding, 511
  UML, 498
SimpleMathController class, **972**–**973**, **976**
SimpleMathPractice class JavaFX style sheet for, **1076**
simple sentinel-controlled loop, 234
simulation of algorithms, 802–807
single-dimensional arrays, 346–410
  accessing elements of, 349–353, 349*t*
  aggregate array operations, 353–366
  changing the array size, 361–362
  comparing arrays for equality, 362–363, 363*f*
  copying arrays, 358–361, 360*f*
  as counters, 401–405
  declaring and instantiating, 346–349
  displaying data as bar chart, 364–366, 364*f*, 365*f*
  examples of (programming activity), 367–370, 396–401
  initial values of array elements, 348*t*
  maximum/minimum values, finding, 357–358, 358*f*
  printing array elements, 354–355, 355*f*
  queues, representing, 733–739
  reading data into arrays, 355–356, 356*f*
  searching arrays
    Binary Search algorithm, 392–396

    - example of (programming activity), 396–401
    - sequential search, 379, 392
  - sorting elements of, 377–401, 381
    - arrays of objects, 391–392
    - Bubble Sort algorithm, 396–399, 399*f*
    - Insertion Sort algorithm, 381, 385–391, 391*f*
    - Selection Sort algorithm, 381–385, 385*f*
  - stacks, representing, 726–730
  - summing array elements, 356–357, 357*f*
  - in user-defined classes, 370–376
- SingleSelectionModel class, **991**
- size method (ArrayList class), 461*t*
- size of ArrayList objects, 459–460
- size of arrays, changing, 361–362
- slash (/)
  - division operator (/), 50, 52*t*, 63*t*
  - shortcut division operator (/=), 61, 62, 63*t*
- slider (JavaFX class), ***968t***
- SliderHandler class, **999**
- sliders, **993–1001**
- Smalltalk, 15, 16
- sorted linked lists, 739–746
  - example of (programming activity), 746–749
- Sorter class
  - with Insertion Sort algorithm, 389
  - with Selection Sort algorithm, 383–384
- sorting algorithms, 802–807
- sorting arrays, 377–401, 381
  - arrays of objects, 391–392
  - Bubble Sort algorithm, 396–399, 399*f*
  - example of (programming activity), 396–401
  - Insertion Sort algorithm, 381, 385–391, 391*f*
  - Selection Sort algorithm, 381–385, 385*f*
- space character, 104*t*
- speed of CPU, 4–5
- Sprite class, 316
- SpriteClient class, 318, 319*f*
- sqrt method (Maths class), 110*t*
- square brackets [ ] for declaring arrays, 347, 427–428
- Square class, in Figure hierarchy, 526–527
- StackOverflowError error, 646–647
- StackPane (JavaFX layout class), ***969t***
- stacks, 643, 646
  - array representation of, 726–730, 727*f*

  defined, 720, 720*f*
  example of (programming activity), 730–733
  implemented with linked lists, 720–722
StackWalker class, 643–644
Stage class, **965**, ***965t***
standard output device, 108
StandardShipping class, 547
start (Application class), ***964t***
start method, 139, **964**, **966**, **1009**
start tag, **966**
StartWithBPredicate test method, 598
statements, 35
static class members, 307–309, 308*t*
  interface members, 537
static class variables, 107
  calling static methods, 106–107
static constants
  Math class, 107, 110*t*, 119
  System class, 107, 108, 108*t*
static data, 107
static keyword, 307, 308
static methods, 106–107, 308, 308*t*
static random access memory (SRAM), 5
Stocks class, **1066**, **1068–1069**
stop (Animation subclasses), ***1025t***, **1034**
stop (Application class), ***964t***
stop (AudioClip class), ***1031t***
storage devices, 4
Stream, 596–601, 596*t*, 599*t*, 644
stream pipeline, 596
String class, 86–88, 95, 249, 253, 391
  constructors, 78*t*
  String literals, 44–45
String comparison methods, 190–192, 190*t*, 192*f*
String concatenation operator, 45
StringDrawing.java, 93–94
StringHandler interface, 539
StringIndexOutOfBoundsException, 92, 567*f*
String methods
  charAt method, 80, 81*t*, 88–89
  demonstrating, 86–87
  indexOf method, 80, 81*t*, 83, 89, 91
  isBlank method, 81*t*, 89
  length method, 81*t*, 82–83, 88

  - strip method, 81*t*, 89
  - substring method, 81*t*, 83, 89, 91–92
  - toLowerCase method, 81*t*, 92
  - toUpperCase method, 81*t*, 83–84, 92
- String objects, after instantiation, 80*f*
- String processing, 90–93
- strings, **1069**
  - comparing, 190–192, 190*t*, 192*f*
  - interfaces and classes, 538, 538*f*
  - palindrome search (example), 654–658
  - parsing with Scanner, 587–589
  - printing backwards, 252–253, 253*f*
  - toString method, 303–306
- strip method (String class), 81*t*, 89
- stroke color, 141
- strokeLine (GraphicsContext class), 143*t*, 146, 151
- stroke method (GraphicsContext class), 151*t*, 152
- strokeOval (GraphicsContext class), 143*t*, 147–148, 151
- strokeRect (GraphicsContext class), 143*t*, 146, 147
- strokeText (GraphicsContext class), 143*t*
- strongly typed language, 38
- structured text files, 587, 593–596
  - example of (programming activity), 593–596
  - parsing string using scanner, 587–589
  - reading structured data using scanner, 589–592
- style sheet, **1075–1077**
- subarrays, 381, 386. *See also* sorting arrays
- subclasses, 496–497. *See also* inheritance
  - constructors for, 502–505, 505*t*
  - inheritance rules, 501*t*, 510*t*, 516–517*t*
  - inheriting from abstract classes, 521–529
  - overriding inherited methods, 508–511
  - specialization, adding, 506–508
  - UML diagrams, 498
- SubHunt class, **897–899**, **1049–1051**
  - UML diagram, **1049*f***
- SubHunt draw method, **902**
- SubHuntViewController class, **899–901**, **1052–1053**
- substring method (String class), 81*t*, 83, 89, 91–92
- subtraction operator (-), 50, 50*t*, 52*t*, 63*t*
- subtract method, BigDecimal class, 187*t*
- summing array elements, 356–357, 357*f*, 438–439, 441–443
- sum of five integers, calculating, 246–247, 247*f*
- superclasses, 496–497. *See also* inheritance

super keyword, 508
  calling superclass methods, 508–510
swapping elements, 383
Swift, 15
switch expression, 195–196
switch statements, 194–199, 237, 238, 322
  example of (programming activity), 200–201
syntax, 2, 34
  arrays
    accessing elements, 349, 431–432
    declaration and instantiation of, 348, 427–428
    method parameter, 448
    method return value, 448
  assignment operator, 49
  conditional operator, 192
  to create JAR file, 25
  declaring variables, 38
  defining classes, 278, 497, 521, 537
  defining constants, 47
  defining interfaces, 537
  do/while loop, 239
  enum objects, 319
  errors, 22, 23
  explicit type casting, 58
  instance variables, 279
  for Javadoc comments, 328
  for JavaFX attribute, **966**–**967**
  methods
    calling, 502, 508
    writing, 280
  return statement, 281
  styling GUI using CSS, **1075**
  switch statements, 194
  try/catch block, 570
  variable number of arguments, 406
  for while loop, 215
System class, 95, 107–109
  System.err method, 573
  System.exit method, 22
  System.in method, 107–109, 108*t*, 573
  System.out, 573
  System.out method, 107–109, 108*t*
system configuration, 7–10
system information

  Mac OS, 7, 9*f*
  Windows, 7, 8*f*

T

tab character, 104*t*
tag section, Javadoc comments, 328–329, 328*t*
tail recursive methods, 651
tail references (queue), 723
Tally class, 449–451
temperature conversion, 64–65
testing applications, 24. *See also* debugging applications
  for loops, 256–257
  if/else statements, 180–182
  linked lists, 705–707
  testing tokens, Scanner methods for, 233*t*
  while loops, 237–238
test plan, 181
text attribute, **974**
text blocks, 46
text fields (JavaFX class), **968*t***, **975*t***
text files, 582–556. *See also* files
  appending data, 582–584
  opening, 573
  reading data from, 219–222, 587–592
  reading structured files, 587–592
  structured files, 593–596
  writing and appending to, 582–556
  writing data to, 608–610
  writing objects to, 608–610
this reference, 299–303
thresholds for floating-point number comparison, 186–187, 187*f*
throw, 568
thumb, **993**
tick marks, **993**
tick values, **993**
TilePuzzle class, **912–915**, **918*f***, **1011–1014**
  UML diagram, **1015*f***
TilePuzzleViewController Class, **918–920**, **918*f***, **1016–1017**
ToDoubleFunction interface, 597*t*
toggleColor method, **983**
toggleGroup property, **978**
toggle variable, 255
toHex method, **1041**
tokens, 104, 105, 249, 587

- counting, 249–251, 252*f*
- testing, Scanner methods for, 233*t*

toLowerCase method (String class), 81*t*, 92

toString method, 407, 464, 466, 468*t*, 475, 568, 705, 739, **1031**
- Array class, 354
- for enum objects, 320*t*, 321
- exception classes, 568
- ShellLinkedList class, 710–712
- user-defined classes, 303–306

touch events, **1047–1061**

TouchPoint class, **1047**

toUpperCase method (String class), 81*t*, 83–84, 92

Towers of Hanoi (example), 665–670

track, **993**

transient variables, 610

Transition class, **1024**

TranslateTransition, **1024**, **1033**

traversal strategies, 825, 826*t*

traversing linked lists, 701

TreasureHunt class, **1055–1057**

treasure hunt game
- PlayTreasureHunt class, **908**
- TreasureHunt class, **904–906**
- TreasureHuntViewController class, **906–908**
- UML diagram, ***904f***

TreasureHuntViewController class, **1058–1059**

tree
- definition, 816
- degree of a node, 817
- depth of a node, 817
- directed edges, 817
- document object model (DOM), 816
- HTML code, 816, 816*f*
- leaf node, 817
- level of a node, 817
- siblings, 817
- storing integers, 817, 817*f*
- terminology, 817–818*t*

trim method (String class), 251, 251*t*

trimToSize method (ArrayList class), 461*t*, 465–466, 696

triple quotes, 46

true (reserved word), 35

truth table
- for DeMorgan's Laws, 167*t*

- for logical operators, 164*t*

try/catch blocks, 567–575
- catching multiple exceptions, 570

tryToPlay method, **1014**

try-with-resources, 573

tweening, **1024**

two-dimensional arrays, 436–456
- accessing elements of, 431–438
- aggregate operations, 436–448
  - displaying data as bar chart, 445–448
  - example of (programming activity), 451–456
  - printing array elements, 437–438
  - processing all elements, 436–438
  - processing columns sequentially, 443–445
  - processing given column, 440–441
  - processing given row, 438–439
  - processing rows sequentially, 441–443
- assigning initial values, 429–431
- declaring and instantiating, 426–431
- elements, accessing, 431–438, 432*t*
- method parameters and return values, 448–451
- processing, 438–439
- visualizing, 432, 432*t*

type casting, 57–59

U

unboxing, 118

unchecked exception, 567, 589

underscore (_) in identifiers, 35

Unicode character set, 13, 14*t*, 35, 191

unicorn character set, C-1–C-2

Unified Modeling Language (UML), 496, 498, 522, 538, 546
- for SubHunt class, **1049*f***
- for TilePuzzle and TilePuzzleViewController classes, **1015*f***

uniform resource identifier (URI), 7

uniform resource locator (URL), 6
- class, 602*t*

unsorted arrays, searching, 378–381

updateAmounts method, **1066**

update method, **1066**, **1073**

update read, 217, 219

URI. *See* uniform resource identifier

URL. *See* uniform resource locator

URLMalformedException exception, 601

useDelimiter method (Scanner class), 587, 587*t*, 588
user-defined classes, 277–330
  accessor methods, 286–289
  arrays in, 370–376
  constructors, 282–286
  data manipulation methods, 293–296
  enumeration types, 318–323
  examples of (programming activity), 323–327
  graphical objects, 316–318
  instance variables, defining, 279–280
  Javadoc, 327–330, 328*t*
  methods, 280–282
  mutator methods, 289–293
  packages, 327
  static class members, 307–309, 308*t*
  this reference, 299–303
  toString and equals methods, 303–306
user-defined exceptions, 617–619
user, reading data from, 217–219

V

validating input, with do/while loop, 240, 240*f*
Validator interface, 543
valueOf method (Double wrapper class), 118*t*
valueOf method (Integer wrapper class), 118*t*
valueOf method, enum objects, 320*t*, 321–323
valueProperty (Slider class), **999*t***
value-returning methods, 80, 281
varargs, 405
variable number of arguments, methods accepting, 405–408
variables, 38. *See also* constants; data types
  class variables, 307
  declaring, 38–39, 41–43
  flag (boolean variables), 250, 260
  instance variables, 76
    default initial values for, 348, 348*t*
    defining, 279–280
    initial values for, 280
    protected, 512–517
  loop control variables, 246–249, 249*f*, 252, 259
  static class, 107
  toggle variable, 255
  transient, 610
VBox (JavaFX layout class), **969, 969*t*, 970, 994, 1024, 1035, 1042–**

1043, 1043*t*
vertical pipe (||) for OR operator, 164, 164*t*, 165*t*, 235
void keyword, as method return type, 80, 81, 281–282
VoteTally class, 450–451

W

web-style documentation, 329, 330*f*
while loops, 222, 257, 394
  event-controlled, 214–215, 217–219
  example of (programming activity), 241–244
  flow of control, 215, 216*f*
  general form for, 215–216
  nesting, 257–261
  syntax, 215
  testing techniques for, 237–238
white space, 35, 104, 104*t*, 249, 250
Windows system information, 7, 8*f*
won method, **1014**, **1018**
World Wide Web Consortium (W3C), 7
wrapper classes, 116
  Character wrapper class, 119–120, 119*t*, 120*f*
  demonstration of, 117, 118*f*
  Double wrapper class, 118–119, 118*t*
  Integer wrapper class, 118, 118*t*
  for primitive data types, 116, 117*t*
wrapper method, 840
WritableImage class, **997*t***
writeObject method (ObjectOutputStream class), 608*t*, 610
Writer class, 576*t*
writing, methods, 280–282

X

XML (Extensible Markup Language), 602, 621, **962**
  special characters, **995*t***
XYChart class, **1069**
XYChart.data constructors, **1072*t***
XYChart.series class, **1072*t***

Y

yield, 195–197

Z

zero, division by, 56–57

zero-iteration loops, 215

The blocks in the diagram are labeled motherboard, input, Output, and Secondary storage. The motherboard contains a C P U with L 1 cache, L 2 cache, and main memory. The C P U with L 1 cache and the L 2 cache have bidirectional data transfer. The L 2 cache and Main memory also have bidirectional data transfer. The input devices are keyboard and L A N. The input devices transmit data to the Motherboard. The output devices are monitor, printer and L A N. The motherboard transfers data to the Output devices. The hard disk is the secondary storage device. The motherboard and the Secondary storage device have bidirectional data transfer between them.

Back to Figure

“The table has 2 columns labelled Item and Value. The row entries are as follows. Row 1. Item: O S Name. Value: Microsoft Windows 11 Home. Row 2. Item: Version. Value: 10 point 0 point 22000 Build 22000. Row 3. Item: Other O S Description. Value: Not Available. Row 4. Item: O S Manufacturer. Value: Microsoft Corporation. Row 5. Item: System Name. Value: DESKTOP hyphen C V B K 8 E Q. Row 6. Item: System Manufacturer. Value: Dell I n c. Row 7. Item: System Model. Value: Inspiron 5515. Row 8. Item: System Type. Value: x 64 hyphen based P C. Row 9. Item: System S K U. Value: 0 A 7 8. Row 10. Item: Processor. Value: A M D Ryzen 7, 5700 U with Radeon Graphics, 1801 M h z, 8 core, left parentheses, s, right parentheses, 16 Logical, dot, dot, dot. Row 11. Item: BIOS Version or Date. Value: Dell I n c, 1 point 4 point 0, 7 forward slash 12 forward slash 2021. Row 12. Item: S M BIOS version. Value: 3 point 3. Row 13. Item: Embedded Controller Version. Value: 255 point 255. Row 14. Item: BIOS mode. Value: U E F I. Row 15. Item: Base Board Manufacturer. Value: Dell I n c. Row 16. Item: Base Board Product. Value: 0 K D K G 8. Row 17. Item: Base Board Version. Value: A 0 0. Row 18. Item: Platform Role. Value: Mobile. Row 19. Item: Secure Boot State. Value: On. Row 20. Item: P C R 7 Configuration. Value: Elevation Required to View. Row 21. Item: Windows Directory. Value: C, colon, back slash, Windows. Row 22. Item: System Directory. Value: C, colon, back slash, Windows, system 32. Row 23. Item: Boot Device. Value: back slash, Device, back slash, Hard disk Volume 1. Row 24. Item: Locale. Value: United States. Row 25. Item: Hardware Abstraction Layer. Value: Version equals double quotes, 10 point 0 point 22000 point 1, double quotes. Row 26. Item: User Name. Value: DESKTOP hyphen C V B K 8 E Q back slash h e r v e. Row 27. Item: Time Zone. Value: Eastern Standard Time. Row 28. Item: Installed Physical Memory, left parentheses, R A, dot, dot, dot. Value: 16.0 G B. Row 29. Item: Total Physical memory. Value: 15.4 G B. Row 30. Item: Available Physical Memory. Value: 7.58 G B. Row 31. Item: Total Virtual Memory. Value: 17.7 G B. Row 32. Item: Available Virtual Memory. Value: 5.90 G B. Row 33. Item: Page File Space. Value: 2.38 G B. Row 34. Item:

Page File. Value: C colon back slash page file dot s y s.”

Back to Figure

The first window has 6 tabs as follows: Overview, Displays, Storage, Memory, Support and Resources. The left half of the window displays the Mac O S logo within a circular region. The system information on the right half of the window are as follows. mac O S Monterey. Version 12.1. i Mac, left parentheses, Retina 5 K, comma, 27 hyphen inch, comma, Late 2015, right parentheses. Processor 3.2 G H z Quad hyphen Core Intel Core i 5. Memory 16 G B 1867 M H z D D R 3. Graphics A M D Radeon R 9 M 380 2 G B. Serial Number D 2 5 S 4 0 Q T G G 7 V. The window has 2 buttons at the bottom right labeled System Report, dot, dot, dot, and Software Update, dot, dot, dot. The caption at the bottom of the window reads: superscript T M, and, Copyright symbol, 1983 hyphen 2021 Apple I n c. All Rights Reserved. License and Warranty. The second window is titled i Mac and it has 2 panes. The left pane has 3 headings labeled Hardware, Network, and Software. The subheadings under Hardware are as follows: A T A, Apple Pay, Audio, Bluetooth, Camera, Card Reader, Controller, Diagnostics, Disc Burning, Ethernet, Fiber Channel, Fire Wire, Graphics, forward slash, Displays, Memory, N V M Express, P C I, Parallel S C S I, Power, Printers, S A S, S A T A, S P I, Storage, Thunderbolt, forward slash, U S B 4, U S B. The subheadings under Network are as follows: Firewall, Locations, Volumes, W W A N, Wi hyphen Fi.

The subheadings under Software are as follows: Accessibility, Applications, Developer, Disabled Software, Extensions. The information displayed on the right half pane are as follows. Hardware Overview, colon. Model Name, colon, i Mac. Model Identifier, colon, i Mac 17 point 1. Processor Name, colon, Quad hyphen Core Intel Core i 5. Processor Speed, colon, 3.2 G H z. Number of Processors, colon, 1. Total Number of Cores, colon, 4. L 2 cache, left parentheses, per Core, right parentheses, colon, 256 K B. L 3 cache, colon, 6 M B. Memory, colon, 16 G B. System Firmware Version, colon, 447 point 60 point 3 point 0 point 0. O S Loader Version, colon, 540 point 60 point 2 hyphen 89. S M C version, left parentheses, system, right parentheses, colon, 2 point 3 3 1 1 2. Serial Number, left parentheses, system, right parentheses, colon, D 2 5 S 4 0 Q T G G 7 V. Hardware U U I D, colon, 3 2 D 1 2 F 6 C hyphen 6 1 D C hyphen 5 F 5 3 hyphen B B B 1 hyphen A 4 E 7 9 0 4 F 3 5 C 0. Provisioning U D I D, colon, 3 2 D 1 2 F 6 C hyphen 6 1 D C hyphen 5 F 5 3 hyphen B B B 1 hyphen A 4 E 7 9 0 4 F 3 5 C 0. The caption at the bottom of the window reads: Kristin's i Mac, right angled bracket, Hardware.

Back to Figure

The folder contains the following 3 files. 1. First Program dot class; CLASS file; 487 bytes. 2. First Program; j GRASP Java file; 247 bytes. 3. Greeting; Executable Jar File; 795 bytes.

Back to Figure

The output is as follows. The area of the circle is 38.4844775.

Back to Figure

The output is as follows. The area of the circle is 1256.636.

Back to Figure

“Line 1: Please choose a language, colon. Line 2, indented once: 1, hyphen, Java. Line 3, indented once: 2, hyphen, Python. Line 4, indented once: 3, hyphen, C plus plus. ”

Back to Figure

“Line 1: The value of constant ZORRO is Z. Line 2: The value of constant PI is 3.14159. Line 3: The number of days in a leap year is 366. Line 4: The number of days in a non hyphen leap year is 365. ”

Back to Figure

“Line 1: The cost of lunch is, dollar sign, 6.84. Line 2: Your age in 2025 is 20. Line 3: The calories in 3 apples is 381. Line 4: The mileage is 28.078947368421055 miles per gallon. ”

Back to Figure

“Line 1: There are 4 quarters in 113 pennies. Line 2: There are 13 pennies left over. Line 3: The monthly salary is 4166.66666666667.”

Back to Figure

“Line 1: The value of result 1 is Infinity. Line 2: The value of result 2 is N a N. Line 3: Exception in thread, double quotes, main, double quotes, java dot lang dot Arithmetic Exception, colon, forward slash, by zero. Line 4, indented once: at Division By Zero dot main, left parentheses, Division By Zero dot java, colon, 15, right parentheses. ”

Back to Figure

“Line 1: At the beginning, a is 6. Line 2: Increment a with prefix notation, colon, 7. Line 3: In the end, a is 7. Line 4: blank. Line 5: At the beginning, b is 2. Line 6: Increment b with postfix notation, colon, 2. Line 7: In the end, b is 3. ”

Back to Figure

“Line 1: a is 5. Line 2: After a plus equals 10, semicolon, a is 15. Line 3: After a minus equals 3, semicolon, a is 12. Line 4: After a asterisk equals 2, semicolon, a is 24. Line 5: After a forward slash equals 6, semicolon, a is 4. Line 6: After a percentage sign, equals, 3, semicolon, a is 1. ”

Back to Figure

In all the 3 diagrams, the object reference is represented by a rectangle and the Object data is represented by 3 rectangles stacked on top of each other. An arrow is drawn from the rectangle representing Object reference to the rectangles representing Object data. In the first diagram, the rectangle representing Object reference is labelled empty String. In the second diagram, the rectangle representing Object reference is labelled greeting. The topmost rectangle of Object data is labelled Hello. In the third diagram, the rectangle representing Object reference is labelled first Name. The topmost rectangle of Object data is labelled Jane.

Back to Figure

The instructions involved in the program are shown on the left and the instructions constituting the method are shown on the right. The flow of control is as follows. Step 1: Program Instruction 1. Step 2: Instruction 2. Step 3: Method Call. The flow of control is now transferred to the method. Step 4: Method instruction 1. Step 5: Method instruction 2. Step 6: Method instruction m. The flow of control now returns to the method call in the Program. Step 7: Program instruction n.

Back to Figure

“Line 1: There are 12 characters. Line 2: The index of uppercase m is 5. Line 3: At index 3, comma, we have a d. Line 4: sub 5 is Morning. Line 5: In caps, colon, GOOD MORNING, all characters of the string good morning are in upper case. Line 6: greeting is still Good Morning. ”

Back to Figure

“Line 1: Null Reference dot java, colon, 10, colon, error, colon, variable greeting might not have been initialized. Line 2, indented once: int len equals greeting dot length, left parentheses, right parentheses, semicolon. Line 3: a caret sign under lowercase g in Line 2. Line 4: 1 error. ”

Back to Figure

“Line 1: Hello has 5 characters. Line 2: Exception in thread, double quotes, main, double quotes, java dot lang dot Null Pointer Exception, colon. Line 3: Cannot invoke, double quotes, String dot length, left parentheses, right parentheses, double quotes, because, double quotes, left angled bracket, local 1, right angled bracket, double quotes, is null. Line 4, indented once: at Null Reference 2 dot main, left parentheses, Null Reference 2 dot java, colon, 15, right parentheses. ”

Back to Figure

The greeting object reference is represented by a rectangle. The caption within the rectangle is null. The Object Data is represented by a rectangle and the caption within the rectangle is Hello.

Back to Figure

“Line 1: s 1 is, colon, O O P in Java. Line 2: s 2 is, colon, is not that difficult. Line 3: s 1 plus s 2 is, colon, O O P in

Java is not that difficult. Line 4: s 1 is still, colon, O O P in Java. Line 5: s 2 is still, colon, is not that difficult. Line 6: blank. Line 7: The length of the empty String is 0. Line 8: blank. Line 9: The first and last characters of Hello are uppercase h and lowercase o. Line 10: blank. Line 11: The index of e is 1. Line 12: The index of l is 2. Line 13: The index of l o is 3. Line 14: The index of h is negative 1. Line 15: blank. Line 16: The middle three characters of Hello are e l l. Line 17: All characters of Hello except the first are e l l o. Line 18: blank. Line 19: Is s 4 blank, question mark, true. Line 20: Is s 4 now empty, question mark, true. ”

Back to Figure

“Line 1: The index of, single quote, comma, single quote, in, double quotes, Lincoln, comma, Abraham, double quotes, is 7. Line 2: The last name is lincoln. Line 3: The first name is abraham. Line 4: blank. Line 5: The formatted full name is Abraham Lincoln. ”

Back to Figure

“Program code. The words in the method name are merged. The code contains the keywords public, and void. Line 28: public void parse Email, left parentheses, right parentheses. Line 29: left brace. Line 30, indented once: forward slash, 5 asterisks, Add your code here, 5 asterisks, forward slash. Line 31, indented once: forward slash, 5 asterisks. 1. Instantiate e mail. Line 32, indented twice: Note, colon, e mail has already been declared. Line 33, indented twice: e mail should contain only one at character. Line 34, indented twice: and only one, period, character. Line 35, indented twice: Example, colon, jane 18 at g mail dot com. Asterisk, forward slash. Line 36: blank. Line 37: blank. Line 38, indented once: forward slash, 5 asterisks. 2. Retrieve user name. Line 39, indented twice: and assign it to the String user Name, semicolon. Line 40, indented twice: Note, colon, user Name has already been declared, asterisk, forward slash. Line 41: blank. Line 42: blank. Line 43, indented once: forward slash, 5 asterisks, 3. Retrieve service provider and. Line 44, indented twice: assign it to the String service Provider, semicolon. Line 45, indented twice: Note, colon, service Provider has already been declared, asterisk, forward slash. Line 46: blank. Line 47: blank. Line 48, indented once: forward slash, 5 asterisks. 4. Retrieve extension and. Line 49, indented twice: assign it to the String extension, semicolon. Line 50, indented twice: Note, colon, extension has already been declared, asterisk, forward slash. Line 51: blank. Line 52: blank. Line 53: right brace. ”

Back to Figure

The window has 4 text labels and 4 text boxes. The text labels are as follows: email, user name, provider, extension. The text boxes are adjacent to the text labels.

Back to Figure

The window has 4 text labels and 4 text boxes. The text labels are as follows: email, user name, provider, extension. The text boxes are adjacent to the text labels. The contents of the text boxes are as follows: jane 18, at symbol, g mail dot com, jane 18, g mail, com.

Back to Figure

“Line 1: The first price is, colon, dollar, 78.67. Line 2: The second price is, colon, dollar 34.50. Line 3: The third price is, colon, dollar 0.33. Line 4: The fourth price is, colon, dollar 3.00. Line 5: The fifth price is, colon, dollar 100.23. Line 6: The first average is, colon, 98.0 dollars. Line 7: The second average is, colon, 98.75 dollars. Line 8: The time is 0 5, colon, 1 2, colon, 0 0. Line 9: millions is 1,234,567. ”

Back to Figure

“Line 1: The die roll is 2. Line 2: The random number between 20 and 200 is 117. ”

Back to Figure

“Line 1: Enter your first name, right angled bracket, Syed. The font color of the string Syed is red. Line 2: Your Name is Syed. Line 3: blank. Line 4: Enter your age as an integer, right angled bracket, 21. The font color of the integer 21 is red. Line 5: Your age is 21. Line 6: blank. Line 7: Enter your G P A, right angled bracket, 3.875. The font color of the real number 3.875 is red. Line 8: Your G

P A is 3.875. ”

Back to Figure

“Line 1: Enter your middle initial, right angled bracket, uppercase A. The font color of character uppercase A is red. Line 2: Your middle initial is uppercase a. ”

Back to Figure

“Line 1: Enter a sentence, right angled bracket, Scanner is useful, period. The font color of the string Scanner is useful, period, is red. Line 2: You said, colon, double quotes, Scanner is useful, period, double quotes. ”

Back to Figure

“Line 1: Enter your first name, right angled bracket, Sarah. The font color of the string Sarah is red. Line 2: Your name is Sarah. Line 3: blank. Line 4: Enter your age as an integer, right angled bracket, lowercase a. The font color of the character lowercase a is red. Line 5: Exception in thread, double quotes, main, double quotes, java dot util dot Input Mismatch Exception. Line 6, indented once: at java dot base, forward slash, java, dot util dot Scanner dot throw For, left parentheses, Scanner dot java, colon, 939, right parentheses. Line 7, indented once: at java dot base, forward slash, java dot util dot Scanner dot next, left parentheses, Scanner dot java, colon, 1594, right parentheses. Line 8, indented once: at java dot base, forward slash, java dot util dot Scanner dot next Int, left parentheses, Scanner dot java, colon, 2258, right parentheses. Line 9, indented once: at java dot base, forward slash, java dot util dot Scanner dot next Int, left parentheses, Scanner dot java, colon, 2212, right parentheses. Line 10, indented once: at Data Input dot main, left parentheses, Data Input dot java, colon, 18, right parentheses. ”

Back to Figure

“Line 1: Combine the arguments using concatenation. Line 2: A double, colon, 23.7, comma, and an int, colon, 78. Line 3: blank. Line 4: Java is case sensitive, colon, lowercase a is different from uppercase a. Line 5: blank. Line 6: Create a variable and print its value. Line 7: The grade is 3.81. ”

Back to Figure

“Line 1: The value of e is 2.718281828459045. Line 2: The value of pi is 3.141592653589793. Line 3: blank. Line 4: The absolute value of 6.7 is 6.7. Line 5: blank. Line 6: The absolute value of negative 6.7 is 6.7. ”

Back to Figure

“Line 1: The log of 5 is 1.6094379124341003. Line 2: The square root of 9 is 3.0. Line 3: 4 to the power of 3 is 64.0. Line 4: 43.5 to the power 3.4 is 372274.65827529586. ”

Back to Figure

“Line 1: 23.4 rounded is 23. Line 2: 23.49 rounded is 23. Line 3: 23.5 rounded is 24. Line 4: 23.51 rounded is 24. Line 5: 23.6 rounded is 24. ”

Back to Figure

“Line 1: The smaller of 8 and 2 is 2. Line 2: The larger of 8 and 2 is 8. Line 3: The smallest of 8, comma, 5, comma, and 12 is 5. ”

Back to Figure

“Line 1: The winning percentage is 68 percent. Line 2: The price is, colon, dollar 78.90. ”

Back to Figure

“Line 1: The int is 42. Line 2: The Integer object is 42 Line 3: Their sum is 84. Line 4: blank. Line 5: The value of i 1 is 76. Line 6: The value of i 2 is 76. Line 7: blank. Line 8: The value of d 1 is 58.32. Line 9: The value of d 2 is 58.32. ”

Back to Figure

“Line 1: Enter a character, right angled bracket, lowercase d. Line 2: The Unicode decimal value of lowercase d is, colon, 100. Line 3: blank. Line 4: lowercase d is a digit, colon, false. Line 5: blank. Line 6: lowercase d is a letter, colon, true. Line 7: lowercase d is uppercase, colon, false. Line 8: lowercase d is lowercase, colon, true. Line 9: blank. Line 10: lowercase d in uppercase is, colon, uppercase d. Line 11: lowercase d is lowercase is, colon, lowercase d. ”

Back to Figure

“Line 1: Enter your first name, right angled bracket, Esmerelda. Line 2: Hello, comma, Esmerelda. Line 3: Your name has 9 letters, period. Line 4: blank. Line 5: In what year were you born, question mark, 2002. Line 6: This year, comma, you will be 18 years old, period. Line 7: So far, comma, you have lived 22.9 percent of your expected life span. Line 8: blank. Line 9: I'm thinking of a number between 1 and 20. Line 10: What is your guess, question mark, 15. Line 11: The secret number was 2. Line 12: Your guess was 13 away, period. ”

Back to Figure

“The output is as follows. Line 1: J a n, tab space, 31. Line 2: F e b, tab space, 24. Line 3: M a r, tab space, 45. Line 4: A p r, tab space, 60. Line 5: May, tab space, 69. Line 6: J u n, tab space, 80. Line 7: J u l, tab space, 88. Line 8: A u g, tab space, 87. Line 9: S e p, tab space, 75. Line 10: O c t, tab space, 65. Line 11: N o v, tab space, 43. Line 12: D e c, tab space, 23. ”

Back to Figure

The data in the bar chart is as follows. J a n: 31. F e b: 24. M a r: 45. A p r: 60. May: 69. J u n: 80. J u l: 88. A u g: 87. S e p: 75. O c t: 65. N o v: 43. D e c: 23. The bars for J a n, Feb, Mar, Nov, and Dec are colored blue. The bars for Apr, May, Sep, and Oct are colored yellow. The bars for Jun, Jul, and Aug are colored red.

Back to Figure

The x axis increases from left to right and the y axis increases from top to bottom. The origin, (0, 0), is at the top left corner. The coordinates at the top right, bottom left and bottom right corners are as follows: (699, 0), (0, 399), and (699, 399). The coordinates marked in the coordinate system are as follows: (75, 60), (150, 250), (350, 250), (550, 300), and (625, 95).

Back to Figure

The first diagram shows a horizontal line. The coordinates of the left end of the line are x Start and y Start. The coordinates of the right end of the line are x End and y End. The method for drawing the line is as follows: stroke Line, left parentheses, x Start, comma, y Start, comma, x End, comma, y End, right parentheses. The second diagram shows a filled rectangle. The horizontal and vertical dimensions of the filled rectangle are labelled Width and Height, respectively. The coordinates of the top left corner of the rectangle are x and y. The method for drawing the filled rectangle is as follows: fill Rect, left parentheses, x, comma, y, comma, width, comma, height, right parentheses. The third diagram shows a filled oval within a dashed rectangle. The edges of the rectangle are tangential to the topmost, bottommost, leftmost, and rightmost points of the oval. The horizontal and vertical dimensions of the dashed rectangle are marked Width and Height, respectively. The method for drawing the filled oval is as follows: fill Oval, left parentheses, x, comma, y, comma, width, comma, height, right parentheses. The fourth diagram shows a caption which is as follows: a string is written here. The coordinates of the leftmost character of the string are x and y. The method for the string is as follows: fill Text, left parentheses, string, comma, x, comma, y, right parentheses.

Back to Figure

The window has 3 lines. A thick green diagonal line runs from the top left to the bottom right. A thick red horizontal line is to the top right of the diagonal line. A thin vertical black line is to the bottom left of the diagonal line.

Back to Figure

The window has 4 shapes. The shape on the left is a blue filled oval within a rectangle. The shape at the center is a filled blue square. The shape at the bottom right is a filled green circle. The shape on the bottom left is an oval.

Back to Figure

The window displays 2 identical strings. The text within the string reads: Programming is not a spectator sport, exclamation mark. The first string has a smaller font size relative to the second string.

Back to Figure

The shapes in the image from top to bottom are as follows: a filled brown rectangle, a horizontal filled line, a vertical red filled oval, a small dark red oval within the larger oval, and 2 filled brown ovals at the bottom of the large oval.

Back to Figure

The table has 2 columns labeled Return Value and Method name and argument list. The row entries are as follows. Row 1. Return Value: void. Method name and argument list: begin Path, left parentheses, right parentheses, sets the drawing path to empty. Row 2. Return Value: void. Method name and argument list: move To, left parentheses, double x, comma, double y, right parentheses, moves to the left parentheses, x, comma, y, right parentheses, location without adding a line to the path. Row 3. Return Value: void. Method name and argument list: line To, left parentheses, double x, comma, double y, right parentheses, adds a line to the current path fro the previous location to left parentheses, x, comma, y, right parentheses. Row 4. Return Value: void. Method name and argument list: close Path, left parentheses, right parentheses, closes the path by adding a line from teh previous location to the first location of the path. Row 5. Return Value: void. Method name and argument list: stroke, left parentheses, right parentheses, draws the path using the current stroke color. Row 6. Return Value: void. Method name and argument list: fill, left parentheses, right parentheses, draws teh path using the current fill color.

Back to Figure

“Line 1: 75, right angled bracket, 18, 2 ampersand signs, 75, left angled bracket, 65 is false. Line 2: 75, left angled bracket, 65, 2 ampersand signs, 75, right angled bracket, 18 is false. Line 3: 75, right angled bracket, 65, 2 pipes, 75, left angled bracket, 18 is true. Line 4: 75, right angled bracket, 65, 2 pipes, 75, left angled bracket, 18, 2 ampersand signs, false is true. Line 5: left parentheses, 75, right angled bracket, 65, 2 pipes, 75, left angled bracket, 18, right parentheses, 2 ampersand signs, false is false. ”

Back to Figure

The steps in the flowchart are as follows. Step 1: condition, question mark. If Step 1 is true, go to step 2, else go to step 3. Step 2: true block. Step 3: next statement.

Back to Figure

“Line 1: Enter your test grade, right angled bracket, 85. Line 2: Enter your extra credit, right angled bracket, 10.

Line 3: Your final test grade is 95. ”

Back to Figure

“Line 1: Enter your test grade, right angled bracket, 85. Line 2: Enter your extra credit, right angled bracket, 0. Line 3: Your final test grade is 85. ”

Back to Figure

The steps in the flowchart are as follows. Step 1: condition, question mark. If Step 1 is true, go to step 2 a, else go to step 2 b. Step 2 a: true block. Step 2 b: false block. Step 3. next statement.

Back to Figure

Line 1: Enter a grade, right angled bracket, 60. Line 2: You passed.

Back to Figure

Line 1: Enter a grade, right angled bracket, 59. Line 2: You failed.

Back to Figure

The steps in the flowchart are as follows. Step 1: condition 1, question mark. If Step 1 is true, go to step 2, else go to step 3. Step 2: true block for condition 1. Step 3: condition 2, question mark. If Step 3 is true, go to step 4 else go to step 5. Step 4: true block for condition 2.

Steps starting from step 5 to step n minus 1 are not shown. Step n: Condition n, question mark. If step n is true, go to step n plus 1, else go to step n plus 2. Step n plus 1: true block for condition n. Step n plus 2: false block, left parentheses, optional, right parentheses. Step n plus 3: next statement. Steps 2, 4, and n pus 1 lead to step n plus 3.

Back to Figure

Line 1: Enter your test grade, colon, 83. Line 2: Your test grade of 83 is a letter grade of uppercase b.

Back to Figure

Line 1: Enter the first integer, colon, 6. Line 2: Enter the second integer, colon, 7. Line 3: Enter the third integer, colon, 5. Line 4: The smallest is 5.

Back to Figure

Line 1: I'm thinking of a number between 1 and 10. What is your guess, question mark, 2. Line 2: The number was 10. Line 3: You missed it by a mile, exclamation mark. Line 4: Better luck next time.

Back to Figure

The window has a text label, a text box, and a button. The text label reads: Enter a golf score. The cursor is displayed as a vertical line within the text box. The button is labeled result.

Back to Figure

The window has a text label, a text box and a button. The text label reads: Thank you. The value displayed within the text box is 73. The button is labeled result. A text below the text label reads: The correct path is. The flow diagram in the window shows the path through which the program instructions are executed. The blocks in the flow diagram are as follows. Block 1: score greater than 72, question mark. true, colon, left. false, colon, right. Blocks 2 and 3 are below block 1. Block 2: true, colon, result equals over par. Block 3: false, colon, score equals equals 72, question mark. true, colon, left. false, colon, right. Diagonal lines are drawn from block 1 to blocks 2 and 3. Blocks 4 and 5 are below block 3. Block 4: true, colon, result equals par. Block 5: false, colon, result equals below par. Diagonal lines are drawn from block 3 to blocks 4 and 5. Block 6: continue. Diagonal lines are drawn from blocks 2, 4, and 5, to block 6. Blocks 1, 2, 6, and the connecting lines between them are shaded green. The text labels at the bottom left of teh window are as follows. The score was 73. The result given by your code is, over par. The result given by your code is, CORRECT. A smiling face is displayed at the top right of the window.

Back to Figure

The window has a text label, a text box and a button. The text label reads: Thank you. The value displayed within the text box is 73. The button is labeled result. A text below the text label reads: The correct path is. The flow diagram in the window shows the path through which the program instructions are executed. The blocks in the flow diagram are as follows. Block 1: score greater than 72, question mark. true, colon, left. false, colon, right. Blocks 2 and 3 are below block 1. Block 2: true, colon, result equals over par. Block 3: false, colon, score equals equals 72, question mark. true, colon, left. false, colon, right. Diagonal lines are drawn from block 1 to blocks 2 and 3. Blocks 4 and 5 are below block 3. Block 4: true, colon, result equals par. Block 5: false, colon, result equals below par. Diagonal lines are drawn from block 3 to blocks 4 and 5. Block 6: continue. Diagonal lines are drawn from blocks 2, 4, and 5, to block 6. Blocks 1, 2, 6, and the connecting lines between them are shaded green. The text labels at the bottom left of teh window are as follows. The score was 73. The result given by your code is, par. The result given by your code is, INCORRECT. A teary eyed sad face is displayed at the top right of the window.

Back to Figure

Line 1: d 1 equals 1.0999999999999999. Line 2: d 2 equals 1.1. Line 3: d 1 and d 2 are not equal. Line 4: blank. Line 5: p i F equals 3.1415927. Line 6: p i d equals 3.141592653589793. Line 7: p i F and p i D are not equal.

Back to Figure

Line 1: d 1 equals 1.0999999999999999. Line 2: d 2 equals 1.1. Line 3: d 1 and d 2 are considered equal. Line 4: blank. Line 5: p i F equals 3.1415927. Line 6: p i d equals 3.141592653589793. Line 7: p i F and p i D are considered equal.

Back to Figure

Line 1: d 1 equals 1.1. Line 2: d 2 equals 1.1. Line 3: d 1 and d 2 are equal.

Back to Figure

The first diagram shows string object s 1. The string object points to the Object Data labeled Hello. The second diagram shows string object s 2. The string object points to the Object Data labeled Hello.

Back to Figure

Line 1: Using, 2 equals sign, comma, s 1 and s 2 are different. Line 2: Using equals, comma, s 1 and s 2 are equal.

Back to Figure

Line 1: Using equals, colon, Green Pastures is not equal to green pastures. Line 2: Using equals Ignore Case, colon, Green Pastures equals green pastures. Line 3: Using compare To, colon, Green Pastures is less than green pastures. Line 4: Using compare To, colon, Green Pastures is less than Green Pastures, roman numeral 2.

Back to Figure

Line 1: Enter an integer, right angled bracket, 12. Line 2: 12 is even.

Back to Figure

Line 1: Enter an integer, right angled bracket, 13. Line 2: 13 is odd.

Back to Figure

Line 1: Welcome to the Calculator. Line 2: Operations are, colon. Line 3: uppercase m, tab space, ADD or plus sign for addition. Line 4: uppercase m, tab space, SUBTRACT or minus sign for subtraction. Line 5: uppercase m, tab space, MULTIPLY or asterisk for multiplication. Line 6: uppercase m, tab space, DIVIDE or forward slash for division. Line 7: Enter your operator, colon, multiply. Line 8: Enter the first operand, colon, 23.4. Line 9: Enter the second operand, colon, 3. Line 10: The result is, colon, 70.2.

Back to Figure

Line 1: Welcome to the Calculator. Line 2: Operations are, colon. Line 3: uppercase m, tab space, ADD or plus sign for addition. Line 4: uppercase m, tab space, SUBTRACT or minus sign for subtraction. Line 5: uppercase m, tab space, MULTIPLY or asterisk for multiplication. Line 6: uppercase m, tab space, DIVIDE or forward slash for division. Line 7: Enter your operator, colon, f. Line 8: Enter the first operand, colon, 52. Line 9: Enter the second operand, colon, 34.5. Line 10: The result is, colon, invalid operation.

Back to Figure

The window has a text label, a text box, and a button. The text label reads: Enter an int. The cursor displayed as a vertical line within the text box. The button is labeled Test.

Back to Figure

The window displays the code path of the switch case statement. The window has a text label, a text box, and a button. The text label reads: Thank you. The value displayed within the text box is 3. The button is labeled Test. The text displayed below the text box is as follows: Value is 3. The switch case construct is represented as a set of connected blocks. The blocks on the left from top to bottom are as follows. Switch, left parentheses, value, right parentheses. case 0, colon, question mark. case 1, colon, question mark. case 2, colon, question mark. case 3, colon, question mark. case 4, colon, question mark. default, colon. A vertical line runs through all the blocks on the left. The blocks in the middle are adjacent to the blocks labeled case and default. The labels on the blocks in the middle from top to bottom are as follows. value is 0. value is 1. value is 2. value is 3. value is 4. value not valid. Horizontal lines run from the blocks on the left to the adjacent blocks in the middle. The block at the bottom right of the window is labeled Continue. Horizontal and vertical lines run from the blocks in the middle to the block at the bottom right. The 5 blocks on the left are shaded green. The fourth block in the middle is shaded green. The block at the bottom right is shaded green. The vertical line running through the 5 blocks on the left is shaded green. The horizontal line running through the fifth block on the left and the fourth block in the middle is shaded green. The horizontal and vertical lines from the fourth block in the middle to the block at the bottom right is shaded green.

Back to Figure

The steps in the flowchart are as follows. Step 1: Initialize variables. Step 2: condition, question mark. If Step 2 is true go to step 3 else go to step 4. Step 3: loop body. Go back to step 2. Step 4: process results.

Back to Figure

Line 1: Enter an integer, comma, or negative 1 to stop, right angled bracket, 23. Line 2: 23. Line 3: Enter an integer, comma, or, negative 1 to stop, right angled bracket, 47. Line 4: 47. Line 5: Enter an integer, comma, or negative 1 to stop, right angled bracket, 100. Line 6: 100. Line 7: Enter an integer, comma, or negative 1 to stop, right angled bracket, negative 1. Line 8: Sentinel value detected. Goodbye.

Back to Figure

The contents of the file are as follows. Line 1: 23. Line 2: 47. Line 3: 100.

Back to Figure

Line 1: 23. Line 2: 47. Line 3: 100. Line 4: End of file detected. Goodbye.

Back to Figure

Line 1: Welcome to the addition calculator. Line 2: blank. Line 3: Enter the first number or 0 for the total, right angled bracket, 34. Line 4: Enter the next number or 0 for the total, right angled bracket, negative 10. Line 5: Enter the next number or 0 for the total, right angled bracket, 2. Line 6: Enter the next number or 0 for the total, right angled bracket, 5. Line 7: Enter the next number or 0 for the total, right angled bracket, 8. Line 8: Enter the next number or 0 for the total, right angled bracket, 0. Line 9: The total is 39.

Back to Figure

Line 1: This program counts the number of passing test scores. Line 2: Enter a negative 1 to stop. Line 3: Enter the first score, right angled bracket, 98. Line 4: Enter the next score, right angled bracket, 75. Line 5: Enter the next score, right angled bracket, 60. Line 6: Enter the next score, right angled bracket, 59. Line 7: Enter the next score, right angled bracket, 45. Line 8: Enter the next score, right angled bracket, 88. Line 9: Enter the next score, right angled bracket, 94. Line 10: Enter the next score, right angled bracket, 96. Line 11: Enter the next score, right angled bracket, 56. Line 12: Enter the next score, right angled bracket, 77. Line 13: Enter the next score, right angled bracket, 82. Line 14: Enter the next score, right angled bracket, 89. Line 15: Enter the next score, right angled bracket, 100. Line 16: Enter the next score, right angled bracket, 78. Line 17: Enter the next score, right angled bracket, 55. Line 18: Enter the next score, right angled bracket, negative 1. Line 19: You entered 15 scores. Line 20: The number of passing test scores is 11. Line 21: 73.3 percent of the class passed the test.

Back to Figure

Line 1: To calculate a class average, comma. Line 2: enter each test score. Line 3: When you are finished, enter a negative 1. Line 4: Enter the first test score, right angled bracket, 88. Line 5: Enter the next test score, right angled bracket, 78. Line 6: Enter the next test score, right angled bracket, 96. Line 7: Enter the next test score, right angled bracket, 75. Line 8: Enter the next test score, right angled bracket, 99. Line 9: Enter the next test score, right angled bracket, 56. Line 10: Enter the next test score, right angled bracket, 78. Line 11: Enter the next test score, right angled bracket, 84. Line 12: Enter the next test score, right angled bracket, 93. Line 13: Enter the next test score, right angled bracket, 79. Line 14: Enter the next test score, right angled bracket, 90. Line 15: Enter the next test score, right angled bracket, 85. Line 16: Enter the next test score, right angled bracket, 79. Line 17: Enter the next test score, right angled bracket, 90. Line 18: Enter the next test score, right angled bracket, 85. Line 19: Enter the next test score, right angled bracket, 79. Line 20: Enter the next test score, right angled bracket, 92. Line 21: Enter the next test score, right angled bracket, 99. Line 22: Enter the next test score, right angled bracket, 94. Line 23: Enter the next test score, right angled bracket, negative 1. Line 24: blank. Line 25: The class average is 85.3.

Back to Figure

Line 1: 88. Line 2: 78. Line 3: 96. Line 4: 75. Line 5: 99. Line 6: 56. Line 7: 78. Line 8: 84. Line 9: 93. Line 10: 79. Line 11: 90. Line 12: 85. Line 13: 79. Line 14: 90. Line 15: 85. Line 16: 79. Line 17: 92. Line 18: 99. Line 19: 94.

Back to Figure

Line 1: This program finds the maximum grade for a class. Line 2: The maximum grade is 99.

Back to Figure

Line 1: Enter your age as an integer, right angled bracket, a. Line 2: Exception in thread, double quotes, main, double quotes, java dot util dot Input Mismatch Exception.

Line 3: at java dot base, forward slash, java dot util dot Scanner dot throw For, left parentheses, Unknown Source, right parentheses. Line 4: at java dot base, forward slash, java dot util dot Scanner dot next, left parentheses, Unknown Source, right parentheses. Line 5: at java dot base, forward slash, java dot util dot Scanner dot next Int, left parentheses, Unknown Source, right parentheses. Line 6: at java dot base, forward slash, java dot util dot Scanner dot next Int, left parentheses, Unknown Source, right parentheses. Line 7: at Read Integer dot main, left parentheses, Read Integer dot java, colon, 13, right parentheses.

Back to Figure

Line 1: Enter your age as an integer, right angled bracket, a s d. Line 2: Please enter an integer, right angled bracket, 12 w g. Line 3: Please enter an integer, right angled bracket, 12.4. Line 4: Please enter an integer, right angled bracket, 23. Line 5: Your age is 23.

Back to Figure

Line 1: Welcome to Bonnie's Burgers. Select from our menu, colon. Line 2, indented multiple times: uppercase c, Cheeseburger, colon, dollar, 7.49. Line 3, indented multiple times: uppercase h, Hot dot, colon, dollar, 6.99. Line 4, indented multiple times: uppercase l, Lemonade, colon, dollar, 2.50. Line 5, indented multiple times: uppercase t, Iced tea, colon, dollar, 2.75. Line 6: Order an item, or, double quotes, uppercase s, double quotes, to stop, right angled bracket, lowercase h. Line 7: Hot dog ordered. Subtotal, colon, dollar, 6.99. Line 8: blank. Line 9: Order another item, or, double quotes, uppercase s, double quotes, to stop, right angled bracket, uppercase t. Line 10: Iced tea ordered. Subtotal, colon, dollar, 9.74. Line 11: blank. Line 12: Order another item, or, double quotes, uppercase s, double quotes, to stop, right angled bracket, lowercase s. Line 13: blank. Line 14: Total order cost is dollar, 9.74.

Back to Figure

The steps in the flowchart are as follows. Step 1: initialize variables. Step 2: loop body. Step 3: condition, question mark. If Step 3 is true, go to Step 2, else go to step 4. Step 4: process results.

Back to Figure

Line 1: Enter a number between 1 and 10, right angled bracket, 20. Line 2: Enter a number between 1 and 10, right angled bracket, negative 1. Line 3: Enter a number between 1 and 10, right angled bracket, 0. Line 4: Enter a number between 1 and 10, right angled bracket, 11. Line 5: Enter a number between 1 and 10, right angled bracket, 5. Line 6: Thank you, exclamation mark.

Back to Figure

The window has an image at the bottom left. The description of the image is as follows: a white rectangle labeled Milk is above a horizontal black line and a brown rectangle labeled Market Foods is at the right end of the horizontal black line. The 3 text labels within the window are as follows. Milk, colon, unit price, equals, dollar sign, 2.00. Your subtotal, equals, dollar sign, 2.00. Correct subtotal, equals, dollar sign, 2.00.

Back to Figure

The steps in the flowchart are as follows. Step 1: initialization. Step 2: condition, question mark. If Step 2 is true, go to step 3 else go to step 5. Step 3: loop body. Step 4: loop update. Go back to step 2. Step 5: process results.

Back to Figure

Line 1: Enter an integer, right angled bracket, 12. Line 2: Enter an integer, right angled bracket, 10. Line 3: Enter an integer, right angled bracket, 5. Line 4: Enter an integer, right angled bracket, 7. Line 5: Enter an integer, right angled bracket, 3. Line 6: The total is 37.

Back to Figure

Line 1: 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20.

Back to Figure

Line 1: Enter integer 1, right angled bracket, 23. Line 2: Enter integer 2, right angled bracket, 12. Line 3: Enter integer 3, right angled bracket, 10. Line 4: Enter integer 4, right angled bracket, 11. Line 5: Enter integer 5, right angled bracket, 15. Line 6: The total is 71.

Back to Figure

Line 1: Enter a sentence, colon. Line 2: Programming is not a spectator, blank spaces, sport, exclamation mark. Line 3: The sentence contains 6 tokens.

Back to Figure

Line 1: Enter a sentence, colon. Line 2: Programming is not a spectator sport, exclamation mark. Line 3: The sentence backwards is, colon. Line 4: exclamation mark, t r o p s, blank space, r o t a t c e p s, blank space, a, blank space, t o n, blank space, s i, blank space, g n i m m a r g o r P.

Back to Figure

The image in the window has 6 alternating and concentric red and black circles. The central circle is red and the outermost circle is black.

Back to Figure

The image shows a filled blue circle circumscribed within a dashed black square. The dimension of the side of the square is marked diameter. The coordinates of the center of the circle are center X and center Y. The radius of the circle is one-half of the diameter. The coordinates of the top left corner of the square are center X minus one-half of the diameter and center Y minus one-half of the diameter.

Back to Figure

Line 1: Enter integer 1, right angled bracket, 23. Line 2: Enter integer 2, right angled bracket, 12. Line 3: Enter integer 3, right angled bracket, 10. Line 4: Enter integer 4, right angled bracket, 11. Line 5: Enter integer 5, right angled bracket, 15. Line 6: The total is 71.

Back to Figure

Line 1: Enter a sentence, colon. Line 2: The sentence backwards is.

Back to Figure

Line 1: 1. Line 2: 1 2. Line 3: 1 2 3. Line 4: 1 2 3 4. Line 5: 1 2 3 4 5.

Back to Figure

Line 1: Enter a positive integer or 0 to exit, right angled bracket, 100. Line 2: Factors of 100, colon, 2, 4, 5, 10, 20, 25, 50. Line 3: blank. Line 4: Enter a positive integer or 0 to exit, right angled bracket, 25. Line 5: Factors of 25, colon, 5. Line 6: blank. Line 7: Enter a positive integer or 0 to exit, right angled bracket, 21. Line 8: Factors of 21, colon, 3, 7. Line 9: blank. Line 10: Enter a positive integer or 0 to exit, right angled bracket, 13. Line 11: Factors of

13, colon, none, comma, 13 is prime. Line 12: blank. Line 13: Enter a positive integer or 0 to exit, right angled bracket, 0.

Back to Figure

The text label at the top is, Enter number of items in cart, colon, 1 to 10. The value within the text box is 5. The button beside the text box is labeled checkout and it is deactivated. The window has an image at its bottom left. The description of the image is as follows. A white rectangle labeled Milk is above a horizontal black line and a brown rectangle labeled Market Foods is at the right end of the horizontal black line. The captions around the image are as follows. Milk, colon, unit price equals, dollar sign, 2.00. Express Lane. Item, hash symbol, 3 of 5. Your subtotal, equals, dollar sign, 9.00. Correct subtotal, equals, dollar sign, 9.00.

Back to Figure

Line 1: Instantiate sedan. Line 2: Instantiate s u v. Line 3: Instantiate mini.

Back to Figure

Line 1: sedan, colon, model is unknown. Line 2, indented once: miles driven is 0. Line 3, indented once: gallons of gas is 0.0. Line 4: s u v, colon, model is Trailblazer. Line 5, indented once: miles driven is 7000. Line 6, indented once: gallons of gas is 437.5.

Back to Figure

Line 1: s u v, colon, model is Trailblazer. Line 2, indented once: miles driven is 7000. Line 3, indented once: gallons of gas is 437.5. Line 4: blank. Line 5: s u v, colon, model is Sportage. Line 6, indented once: miles driven is 200. Line 7, indented once: gallons of gas is 10.5. Line 8: blank. Line 9: s u v, colon, model is Sportage. Line 10, indented once: miles driven is 200. Line 11, indented once: gallons of gas is 10.5.

Back to Figure

Line 1: Mileage for s u v is 16.0. Line 2: Gas cost for s u v is 1,220.62 dollars.

Back to Figure

The first text label reads: airport 1. The caption within the text box below the label reads: null. The second text label reads: airport 2. The caption within the text box below the label reads: null. the captions at the bottom of the window are as follows: Two airport object references declared, colon. Airport airport 1, comma, airport 2, semicolon.

Back to Figure

The window has 2 images representing the objects airport 1 and airport 2. Each object is represented by a rectangle. Each object has 2 attributes represented by rectangles. The attributes and the associated values of the airport 1 object are as follows. airport code: I A D, gates: 30. The attributes and the associated values of the airport 2 object are as follows. airport code: J F K, gates: 69. The caption at the bottom of the window reads: Actions are complete.

Back to Figure

Line 1: car is Auto, at symbol, 4 a 5 7 4 7 9 5. Line 2: this is Auto, at symbol, 4 a 5 7 4 7 9 5. Line 3: Miles driven is 67. Line 4: Gallons of gas is 3.4.

Back to Figure

Line 1: Model, colon, Spyder, semicolon, miles driven, colon, 0, semicolon, gallons of gas, colon, 0.00. Line 2: Model, colon, Accent, semicolon, miles driven, colon, 0, semicolon, gallons of gas, colon, 0.00. Line 3: sporty and compact are not equal.

Back to Figure

The User is represented by a humanoid figure. Rectangles represent Client and Class. An arrow from the User to the Client is marked Sends requests. An arrow from the Client to the User is marked Displays results. An arrow from the Client to the Class is marked Calls methods. An arrow from the Class to the Client is marked Returns values.

Back to Figure

Line 1: Before instantiating an Auto object, colon. Line 2:

the count of Auto objects is 0. Line 3: blank. Line 4: After instantiating an Auto object, colon. Line 5: the count of Auto objects is 1.

Back to Figure

Line 1: r 1 equals 5, forward slash, 3. Line 2: r 2 equals 7. Line 3: r 3 equals 0. Line 4: r 4 equals negative 5 forward slash 7. Line 5: g c d, left parentheses, 1635, comma, 75, right parentheses, equals, 15. Line 6: g c d, left parentheses, 7 asterisk 11 asterisk 13, comma, 3 asterisk 7 asterisk 11, right parentheses, equals, 77. Line 7: r 5 equals 25, forward slash, 10. Line 8: In lowest terms, comma, r 5 equals 5, forward slash, 2. Line 9: r 6 equals negative 2310, forward slash, 1771. Line 10: In lowest terms, comma, r 6 equals negative 30, forward slash, 23. Line 11: 5, forward slash, 3, plus, 6, forward slash, 4, equals, 19, forward slash, 6. Line 12: 5, forward slash, 3, asterisk, 6, forward slash, 4, equals, 5, forward slash, 2.

Back to Figure

The window has 3 sprites at the top left, center and bottom right. Each sprite consists of a vertical oval and 2 horizontal ovals at the bottom, a small oval within the large vertical oval and a hat like structure which has a rectangle and a horizontal line below.

Back to Figure

Line 1: Comparing objects using equals. Line 2: W E D does not equal F R I. Line 3: blank. Line 4: Comparing Objects using compare To. Line 5: W E D is less than F R I. Line 6: blank. Line 7: Getting the ordinal value. Line 8: The value of W E D is 3. Line 9: blank. Line 10: Converting a String to an object. Line 11: The value of day is M O N.

Back to Figure

Line 1: Enter a day. Line 2: left parentheses, Sun, comma, M o n, comma, T u e, comma, W e d, comma, T h u r, comma, F r i, comma, S a t, right parentheses, right angled bracket, F r i. Line 3: The special for F R I is fish fillet.

Back to Figure

The window has 3 text labels and 3 text boxes. The first text label reads, count Airports. The value within the text box is 0. The second text label reads, airport 1. The value within the text box is null. The third text label reads, airport 2. The value within the text box reads, null. The captions at the bottom of the window are as follows: Two airport object references declared, colon. Airport airport 1, comma, airport 2, semicolon.

Back to Figure

The mutator method is set Miles Driven and the accessor method is get Miles driven. The description is listed below each method.

Back to Figure

The variable cell Bills is represented by a rectangle. The array elements are represented by 6 rectangles stacked atop each other. An arrow is drawn from the variable to the topmost rectangle of the array elements. The labels and corresponding values of the 6 stacked rectangles are as follows. cell bills, left square bracket, 0, right square bracket: 0.00. cell bills, left square bracket, 1, right square bracket: 0.00. cell bills, left square bracket, 2, right square bracket: 0.00. cell bills, left square bracket, 3, right square bracket: 0.00. cell bills, left square bracket, 4, right square bracket: 0.00. cell bills, left square bracket, 5, right square bracket: 0.00.

Back to Figure

The variable cell Bills is represented by a rectangle. The array elements are represented by 6 rectangles stacked atop each other. An arrow is drawn from the variable to the topmost rectangle of the array elements. The labels and corresponding values of the 6 stacked rectangles are as follows. cell bills, left square bracket, 0, right square bracket: 45.24. cell bills, left square bracket, 1, right square bracket: 54.67. cell bills, left square bracket, 2, right square bracket: 42.55. cell bills, left square bracket, 3, right square bracket: 44.61. cell bills, left square bracket, 4, right square bracket: 65.29. cell bills, left square bracket, 5, right square bracket: 49.75.

Back to Figure

Line 1: The first monthly cell bill is 45.24. Line 2: The last monthly cell bill is 49.75.

Back to Figure

Line 1: cars, left square bracket, 0, right square bracket, is a Ferrari. Line 2: my Car has used 0.0 gallons of gas. Line 3: Exception in thread, double quotes, main, double quotes, java dot lang dot Null Pointer Exception, colon. Line 4, indented once: Cannot invoke, double quotes, Auto dot get Model, left parentheses, right parentheses, double quotes, because, double quotes, left angled bracket, local 1, right angled bracket, left square bracket, 2, right square bracket, double quotes, is null. Line 5, indented once: at Auto Array dot main, left parentheses, Auto Array dot java, colon, 26, right parentheses.

Back to Figure

Line 1: Element. Tab space. Value. Line 2: 0. Tab space. 45.24. Line 3: 1. Tab space. 54.67. Line 4: 2. Tab space. 42.55. Line 5: 3. Tab space. 44.61. Line 6: 4. Tab space. 65.29. Line 7: 5. Tab space. 49.75.

Back to Figure

Line 1: Enter bill amount for month 1. 63.33. Line 2: Enter bill amount for month 2. 54.27. Line 3: Enter bill amount for month 3. 71.19. Line 4: Enter bill amount for month 4.

59.03. Line 5: Enter bill amount for month 5. 62.65. Line 6: Enter bill amount for month 6. 65.08.

Back to Figure

Total for the bills, colon, dollar sign, 302.11.

Back to Figure

The highest bill, comma, dollar sign, 65.29, comma, was found at index 4.

Back to Figure

The contents of the elements of cell Bills array are as follows. cell Bills, left square bracket, 0, right square bracket: 45.24. cell Bills, left square bracket, 1, right square bracket: 54.67. cell Bills, left square bracket, 2, right square bracket: 42.55. cell Bills, left square bracket, 3, right square bracket: 44.61.cell Bills, left square bracket, 4, right square bracket: 65.29. cell Bills, left square bracket, 5, right square bracket: 49.75. The contents of the elements of bills Backup array are as follows. bills Backup, left square bracket, 0, right square bracket: 0.00. bills Backup, left square bracket, 1, right square bracket: 0.00. bills Backup, left square bracket, 2, right square bracket: 0.00. bills Backup, left square bracket, 3, right square bracket: 0.00. bills Backup, left square bracket, 4, right square bracket: 0.00. bills Backup, left square bracket, 5, right square bracket: 0.00.

Back to Figure

Before the assignment, bills Backup pointed to 6 memory locations. The value in each memory location is 0.00. After the assignment, cell Bills and bills Backup arrays point to 6 memory locations which are as follows. bills Backup, left square bracket, 0, right square bracket, right arrow, cell Bills, left square bracket, 0, right square bracket: 45.24. bills Backup, left square bracket, 1, right square bracket, right arrow, cell Bills, left square bracket, 1, right square bracket: 54.67. bills Backup, left square bracket, 2, right square bracket, right arrow, cell Bills, left square bracket, 2, right square bracket: 42.55. bills Backup, left square bracket, 3, right square bracket, right arrow, cell Bills, left square bracket, 3, right square bracket: 44.61. bills Backup, left square bracket, 4, right square bracket, right arrow, cell Bills, left square bracket, 4, right square bracket: 65.29. bills Backup, left square bracket, 5, right square bracket, right arrow, cell Bills, left square bracket, 5, right square bracket: 49.75.

Back to Figure

Both variable names point to the same set of 6 memory locations. The contents of the memory locations are as follows. bills Backup, left square bracket, 0, right square bracket, right arrow, cell Bills, left square bracket, 0, right square bracket: 45.24. bills Backup, left square bracket, 1, right square bracket, right arrow, cell Bills, left square bracket, 1, right square bracket: 54.67. bills Backup, left square bracket, 2, right square bracket, right arrow, cell Bills, left square bracket, 2, right square bracket: 42.55. bills Backup, left square bracket, 3, right square bracket, right arrow, cell Bills, left square bracket, 3, right square bracket: 44.61. bills Backup, left square bracket, 4, right square bracket, right arrow, cell Bills, left square bracket, 4, right square bracket: 38.00. bills Backup, left square bracket, 5, right square bracket, right arrow, cell Bills, left square bracket, 5, right square bracket: 49.75.

Back to Figure

The cell Bills array points to 6 consecutive memory locations. The contents of the memory locations are as follows. cell Bills, left square bracket, 0, right square bracket: 45.24. cell Bills, left square bracket, 1, right square bracket: 54.67. cell Bills, left square bracket, 2, right square bracket: 42.55. cell Bills, left square bracket, 3, right square bracket: 44.61. cell Bills, left square bracket, 4, right square bracket: 65.29. cell Bills, left square bracket, 5, right square bracket: 49.75. The bills Backup array points to 6 consecutive memory locations. The contents of the memory locations are as follows. bills Backup, left square bracket, 0, right square bracket: 45.24. bills Backup, left square bracket, 1, right square bracket: 54.67. bills Backup, left square bracket, 2, right square bracket: 42.55. bills Backup, left square bracket, 3, right square bracket: 44.61. bills Backup, left square bracket, 4, right square bracket: 65.29. bills Backup, left square bracket, 5, right square bracket: 49.75.

Back to Figure

Line 1: cell Bills. Line 2: Element, tab space, Value. Line 3: 0, tab space, 45.24. Line 4: 1, tab space, 54.67. Line 5: 2, tab space, 42.55. Line 6: 3, tab space, 44.61. Line 7: 4, tab space, 65.29. Line 8: 5, tab space, 49.75. Line 9: blank. Line 10: bills Backup. Line 11: Element, tab space, Value. Line 12: 0, tab space, 45.24. Line 13: 1, tab space, 54.67. Line 14: 2, tab space, 42.55. Line 15: 3, tab space, 44.61. Line 16: 4, tab space, 38.0. Line 17: 5, tab space, 49.75.

Back to Figure

cell Bills 1 and cell Bills 2 are not equal.

Back to Figure

The window displays a bar chart. The height of the bars from left to right are as follows: 45.24, 54.67, 42.55, 44.61, 65.29, and 49.75.

Back to Figure

The diagram shows a rectangle whose height is equal to cell Bins, left square bracket, i, right square bracket and whose width is equal to BAR underscore WIDTH. The height is much larger than the width. The x and y coordinates of the top left corner of the rectangle are x Start, and BASE underscore Y underscore BAR minus cell Bills, left square bracket, i, right square bracket, respectively. The base of the rectangle is at BASE underscore Y underscore BAR. The BASE underscore Y underscore VALUE is cell Bills, left square bracket, i, right square bracket.

Back to Figure

The window has 5 deactivated buttons arranged in 2 rows. The labels of the 3 buttons in the first row are as follows: Fill Values, Print Array, Set Values. The labels of the 2 buttons in the second row are as follows: Count Frequency, Find Minimum. The captions below the buttons are as follows. Your current frequency count, colon, 1. Correct current frequency count, colon, 1. A bar graph is displayed at the bottom of the window. The height of the bars from left to right are as follows: 73, 57, 67, 56, 69, 73, 69, 68, 57, 67, 75, 59, 67, 58, and 74. The leftmost bar is colored red. The fifth bar is colored blue and it has a red outline. All the other bars are colored blue.

Back to Figure

The window has 5 buttons arranged in 2 rows. The labels of the 3 buttons in the first row are as follows: Fill Values, Print Array, Set Values. The labels of the 2 buttons in the second row are as follows: Count Frequency, Find Minimum. A bar graph is displayed at the bottom of the window. The height of the bars from left to right are as follows: 73, 57, 67, 56, 69, 73, 69, 68, 57, 67, 75, 59, 67, 58, and 74.

Back to Figure

Line 1: c 1, equals. Line 2: dollar sign, 0.00, blank space, dollar sign, 0.00, blank space, dollar sign, 0.00, blank space, dollar sign, 0.00, blank space, dollar sign, 0.00, blank space, dollar sign, 0.00. Line 3: blank. Line 4: c 2 equals 555 minus 555 minus 5555. Line 5: dollar 24.60, blank space, dollar sign, 48.75, dollar sign, 62.50. Line 6: blank. Line 7: The highest bill is dollar sign, 62.50. Line 8: blank. Line 9: The total of all bills is dollar sign, 135.85. Line 10: blank. Line 11: c 2 and c 3 are equal. Line 12: blank. Line 13: after client changes original array. Line 14: c 2 equals 555 minus 555 minus 5555. Line 15: dollar sign, 24.60, blank space, dollar sign, 48.75, blank space, dollar sign, 62.50. Line 16: blank. Line 17: after client changes returned array. Line 18: c 2, equals, 555 minus 555 minus 5555. Line 19: dollar sign 24.60, blank space, dollar sign, 48.75, blank space, dollar sign, 62.50.

Back to Figure

Line 1: The number of parameters is 2. Line 2: a r g s, left square bracket, 0, right square bracket, colon, input dot t x t. Line 3: a r g s, left square bracket, 1, right square bracket, colon, output dot t x t.

Back to Figure

Line 1: Enter the member's I D or 0 to stop, right angled bracket, 1234. Line 2: Sorry, comma, member is not a winner. Line 3: blank. Line 4: Enter the next member's I D or 0 to stop, right angled bracket, 3980. Line 5: You win free popcorn, exclamation mark. Line 6: blank. Line 7: Enter the next member's I D or 0 to stop, right angled bracket, 0. Line 8: blank. Line 9: The winners were 619, blank space, 4510, blank space, 3980, blank space, 4004.

Back to Figure

Line 1: Before Selection Sort, comma, the array is. Line 2: 3394, blank spaces, 279, blank spaces, 1181, blank spaces, 2471, blank spaces, 3660, blank spaces, 221. Line 3: blank. Line 4: After Selection Sort, comma, the array is. Line 5: 221, blank spaces, 279, blank spaces, 1181, blank spaces, 2471, blank spaces, 3394, blank spaces, 3660.

Back to Figure

The 3 yellow cards on the left are numbered 3, 5, and 9. The 2 white cards on the right are numbered 2 and 8. A white card numbered 4 is inserted between the yellow card numbered 9 and the white card numbered 2.

Back to Figure

The 2 yellow cards on the left are numbered 3 and 5. The yellow card in the middle is numbered 9. The 2 white cards on the right are numbered 2 and 8. A white card numbered 4 is inserted between yellow cards numbered 5 and 9. A question mark is marked above the white card numbered 4.

Back to Figure

The yellow card on the left is numbered 3. The 2 yellow cards in the middle are numbered 5 and 9. The 2 white cards on the right are numbered 2 and 8. A white card numbered 4 is inserted between the yellow cards labeled 3 and 5. A question mark is marked above the white card numbered 4.

Back to Figure

The 4 yellow cards from left to right are numbered 3, 4, 5, and 9. The 2 white cards on the right are numbered 2 and 8.

Back to Figure

Line 1: Before Insertion Sort, comma, the array is. Line 2: 2856, tab space, 2384, blank spaces, 3979, blank spaces, 3088, blank spaces, 1176, blank spaces, 284. Line 3: blank. Line 4: After Insertion Sort, comma, the array is. Line 5: 284, blank spaces, 1176, blank spaces, 2384, blank spaces, 2856, blank spaces, 3088, blank spaces, 3979.

Back to Figure

Line 1: Enter a value to search for, right angled bracket, 64. Line 2: 64 found at index 12.

Back to Figure

The window has 2 deactivated buttons labeled Sequential Search and Bubble Sort. The caption below the buttons reads: Sorting using Bubble Sort. A bar graph is displayed at the bottom of the window. The heights of the bars from left to right are as follows: 50, 60, 59, 56, 64, 65, 62, 67, 70, 70, 69, 74, 75, 78, 80. The bars corresponding to 56 and 64 are shaded red and all the other bars are labeled blue.

Back to Figure

The window has 2 buttons labeled Sequential Search and Bubble Sort. A bar graph is displayed at the bottom of the window. The heights of the bars from left to right are as follows: 80, 60, 65, 75, 50, 78, 64, 59, 56, 74, 62, 67, 70, 70, and 69.

Back to Figure

Line 1: Roll. Tab space. Count. Line 2: 1. Tab space. 81. Line 3: 2. Tab space. 82. Line 4: 3. Tab space. 89. Line 5: 4. Tab space. 78. Line 6: 5. Tab space. 87. Line 7: 6. Tab space. 83.

Back to Figure

Line 1: Mary, colon, rating, 4.0. Line 2: Mary, colon, rating, 4.7. Line 3: Mary, colon, rating, 4.5. Line 4: Mary, colon, rating, 4.5.

Back to Figure

The table has 7 columns labeled Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday. The rows are labeled Week 1, Week 2, and so on, up to Week 51 and Week 52. The row entries are as follows. Row 1. Week 1. Sunday: 35. Monday: 28.6. Tuesday: 29.3. Wednesday: 38. Thursday: 43.1. Friday: 45.6. Saturday: 49. Row 2. Week 2. Sunday: 51.9. Monday: 37.9. Tuesday: 34.1. Wednesday: 37.1. Thursday: 39. Friday: 40.5. Saturday: 43.2. Rows from row 3 to row 50 are blank. Row 51. Week 51. Sunday: 56.2. Monday: 51.9. Tuesday: 45.3. Wednesday: 48.7. Thursday: 42.9. Friday: 35.5. Saturday: 38.2. Row 52. Week 52. Sunday: 33.2. Monday: 27.1. Tuesday: 24.9. Wednesday: 29.8. Thursday: 37.7. Friday: 39.9. Saturday: 38.8.

Back to Figure

The variable numbers List 1 points to row 0 of the array. Row 0 points to the first of 3 elements of a row whose indices and values are as follows. Index: left square bracket, 0, right square bracket, left square bracket, 0, right square bracket. Value: 0. Index: left square bracket, 0, right square bracket, left square bracket, 1, right square bracket. Value: 5. Index: left square bracket, 0, right square bracket, left square bracket, 2, right square bracket. Value: 10. Row 1 points to the first of 4 elements of a row whose indices and values are as follows. Index: left square bracket, 1, right square bracket, left square bracket, 0, right square bracket. Value: 0. Index: left square bracket, 1, right square bracket, left square bracket, 1, right square bracket. Value: 3. Index: left square bracket, 1, right square bracket, left square bracket, 2, right square bracket. Value: 6. Index: left square bracket, 1, right square bracket, left square bracket, 3, right square bracket. Value: 9.

Back to Figure

The variable family Cell Bills points to row 0 of the 3 row array. Row 0 points to the first element of a 4 element row whose indices and values are as follows. Index: left square bracket, 0, right square bracket, left square bracket, 0, right square bracket. Value: 45.24. Index: left square bracket, 0, right square bracket, left square bracket, 1, right square bracket. Value: 54.67. Index: left square bracket, 0, right square bracket, left square bracket, 2, right square bracket. Value: 32.55. Index: left square bracket, 0, right square bracket, left square bracket, 3, right square bracket. Value: 25.61. Row 1 points to the first element of a 4 element row whose indices and values are as follows. Index: left square bracket, 1, right square bracket, left square bracket, 0, right square bracket. Value: 65.29. Index: left square bracket, 1, right square bracket, left square bracket, 1, right square bracket. Value: 49.75. Index: left square bracket, 1, right square bracket, left square bracket, 2, right square bracket. Value: 32.08. Index: left square bracket, 1, right square bracket, left square bracket, 3, right square bracket. Value: 26.11. Row 2 points to the first element of a 4 element row whose indices and values are as follows. Index: left square bracket, 2, right square bracket, left square bracket, 0, right square bracket. Value: 75.24. Index: left square bracket, 2, right square bracket, left square bracket, 1, right square bracket: 54.53. Index: left square bracket, 2, right square bracket, left square bracket, 2, right square bracket: 34.55. Index: left square bracket, 2, right square bracket, left square bracket, 3, right square bracket. Value: 28.16.

Back to Figure

Line 1: The first monthly cell bill for the first family member is 45.24. Line 2: The last monthly cell bill for the last family member is 28.16. Line 3: blank. Line 4: The number of rows is 3. Line 5: The number of columns in row 0 is 4. Line 6: The number of columns in row 1 is 4. Line 7: The number of columns in row 2 is 4.

Back to Figure

Line 1: cars, left square bracket, 1, right square bracket, left square bracket, 0, right square bracket, single quote, s, description is, colon. Line 2: Model, colon, Ferrari, semicolon, miles driven, semicolon, 0, semicolon, gallons of gas, colon, 0.0.

Back to Figure

Line 1: Tab space. Data for family cell bills. Line 2: row 0, colon. Tab space. 45.24. Tab space. 54.67. Tab space. 32.55. Tab space. 25.61. Line 3: row 1, colon. Tab space. 65.29. Tab space. 49.75. Tab space. 32.08. Tab space. 26.11. Line 4: row 2, colon. Tab space. 75.24. Tab space. 54.53. Tab space. 34.55. Tab space. 28.16.

Back to Figure

Line 1: Month 0, colon, July. Line 2: Month 1, colon, August. Line 3: Month 2, colon, September. Line 4: Enter a month number between 0 and 2, right angled bracket, 1. Line 5: The total family cell bills during August is dollar sign, 173.23.

Back to Figure

Line 1: Family member 0, colon, Joe. Line 2: Family member 1, colon, Jane. Line 3: Family member 2, colon, Mike. Line 4: Family member 3, colon, Sarah. Line 5: Enter a family member between 0 and 3, right angled bracket, 2. Line 6: blank. Line 7: The max cell bill for Mike is dollar sign, 34.55.

Back to Figure

Line 1: The total for July is, dollar sign, 158.07. Line 2: The total for August is, dollar sign, 173.23. Line 3: The total for September is, dollar sign, 192.48.

Back to Figure

Line 1: The maximum number of columns in grades is 4.

Line 2: blank. Line 3: Column 0, colon, 89. Tab space. 84. Tab space. 80. Line 4: Column 1, colon, 75. Tab space. 76. Tab space. 88. Line 5: Column 2, colon. Tab space. 92. Tab space. 95. Line 6: Column 3, colon. Tab space. 96.

Back to Figure

The window displays 3 bar graphs labeled July, August, and September. The data corresponding to July is as follows: 45.24, 54.67, 32.55, and 25.61. The data corresponding to August is as follows: 65.29, 49.75, 32.08, 26.11. The data corresponding to September is as follows: 75.24, 54.53, 34.55, 28.16.

Back to Figure

Line 1: Total votes per candidate. Line 2: Smith. Tab space. Jones. Tab space. Berry. Tab space. Chase. Line 3: 1543. Tab space. 2615. Tab space. 1845. Tab space. 2534. Line 4: blank. Line 5: Total votes per district. Line 5: 1. Tab space. 2. Tab space. 3. Tab space. 4. Tab space. 5. Tab space. 6. Line 6: 873. Tab space. 1955. Tab space. 1156. Tab space. 896. Tab space. 2185. Tab space. 1472.

Back to Figure

The window has 5 deactivated buttons whose labels are as follows: Fill Values, Print Array, Set Values, Count Frequency, and Find Minimum. The caption below the buttons is as follows: Your current frequency count for 61, colon, 4. The window displays 4 bar graphs labeled Row 0, Row 1, Row 2, and Row 3. The data corresponding to Row 0 is as follows: 76, 80, 59, 66, 60, 71, 61, 50, 60, 78, 50, 60, 66, 77, 80, 67, 77, 80, 67, 71, 52, 61, 61. The bars corresponding to 61 are colored red and all the other bars are colored blue. The data corresponding to Row 1 is as follows: 77, 80, 80, 70, 70, 73, 59, 78, 56, 71, 56, 55, 62, 70, 61, 73, 50, 56, 75, 52. The bar corresponding to 61 is colored red and all the other bars are colored green. The data corresponding to Row 2 is as follows: 58, 66, 51, 54, 52, 60, 59, 78, 56, 64, 79, 60, 73, 78, 72, 53, 59, 50, 55, 66. All the bars are colored gray. The bar corresponding to 54 has a red outline. The data corresponding to Row 3 is as follows: 75, 76, 55, 68, 56, 59, 62, 59, 60, 76, 50, 70, 50, 72, 58, 77, 55, 59, 56, 54. All the bars are colored purple.

Back to Figure

The window has 5 buttons whose labels are as follows: Fill Values, Print Array, Set Values, Count Frequency, and Find Minimum. The window displays 4 bar graphs labeled Row 0, Row 1, Row 2, and Row 3. The data corresponding to Row 0 is as follows: 76, 80, 59, 66, 60, 71, 61, 50, 60, 78, 50, 60, 66, 77, 80, 67, 77, 80, 67, 71, 52, 61, 61. All the bars are colored blue. The data corresponding to Row 1 is as follows: 77, 80, 80, 70, 70, 73, 59, 78, 56, 71, 56, 55, 62, 70, 61, 73, 50, 56, 75, 52. All the bars are colored green. The data corresponding to

Row 2 is as follows: 58, 66, 51, 54, 52, 60, 59, 78, 56, 64, 79, 60, 73, 78, 72, 53, 59, 50, 55, 66. All the bars are colored gray. The data corresponding to Row 3 is as follows: 75, 76, 55, 68, 56, 59, 62, 59, 60, 76, 50, 70, 50, 72, 58, 77, 55, 59, 56, 54. All the bars are colored purple.

Back to Figure

Line 1: Using the traditional for loop, colon. Line 2: 34. Tab space. 89. Tab space. 65. Line 3: Using the enhanced for loop, colon. Line 4: 34. Tab space. 89. Tab space. 65. Line 5: Using unboxing and enhanced for loop, colon. Line 6: 34. Tab space. 89. Tab space. 65. Line 7: After calling set, left parentheses, 1, comma, 100, right parentheses, semicolon. Line 8: 34. Tab space. 100. Tab space. 65. Line 9: At index 0, comma, 34 was removed. Line 10: After removing the element at index 0, colon. Line 11: 100. Tab space. 65.

Back to Figure

Line 1: Enter a keyword, right angled bracket, Java. Line 2: Our book collection is, colon. Line 3: title, colon, Intro to Java. Tab space. author, colon, James. Tab space. price, colon, 56.99. Line 4: title, colon, Advanced Java. Tab space. author, colon, Green. Tab space. price, colon, 65.99. Line 5: title, colon, Java Servlets. Tab space. author, colon, Brown. Tab space. price, colon, 75.99. Line 6: title, colon, Intro to H T M L. Tab space. author, colon, James. Tab space. price, colon, 29.49. Line 7: title, colon, Intro to Flash. Tab space. author, colon, James. Tab space. price, colon, 34.99. Line 8: title, colon, Advanced H T M L. Tab space. author, colon, Green. Tab space. price, colon, 56.99. Line 9: blank. Line 10: Enter a keyword, right angled bracket, Java. Line 11: The search results for Java are, colon. Line 12: title, colon, Intro to Java. Tab space. author, colon, James. Tab space. price, colon, 56.99. Line 13: title, colon, Advanced Java. Tab space. author, colon, Green. Tab space. price, colon, 65.99. Line 14: title, colon, Java Servlets. Tab space. author, colon, Brown. Tab space. price, colon, 75.99.

Back to Figure

The window has 5 buttons whose labels are as follows: Fill Cars, Print Auto List, Set Model, Count Model Frequency, and Find Maximum Miles. The window has an image of a car labeled B M W. The captions below the image are as follows. Miles equals 4000. Gallons equals 200.0. Student's current count is 2. Correct current count is 2.

Back to Figure

The window has 5 buttons whose labels are as follows: Fill Cars, Print Auto List, Set Model, Count Model Frequency, and Find Maximum Miles.

Back to Figure

Line 1: States, colon, left brace, M D, equals, Maine, comma, N Y, equals, New York, comma, N J, equals, New Jersey, comma, C A, equals, California, right brace. Line 2: N Y is New York. Line 3: blank. Line 4: states, colon, left brace, M D, equals, Maryland, comma, N Y, equals, New York, comma, N J, equals, New Jersey, comma, C A, equals, California, right brace. Line 5: Maryland was removed. Line 6: blank. Line 7: states, colon, left brace, N Y, equals, New York, comma, N J, equals, New Jersey, comma, C A, equals, California, right brace. Line 8: blank. Line 9: N Y, colon, New York. Line 10: N J, colon, New Jersey. Line 11: C A, colon, California. Line 12: blank. Line 13: New York. Line 14: New Jersey. Line 15: California.

Back to Figure

The diagram shows 5 classes which are labeled Vehicle, Automobile, Truck, Pickup, and Tractor Trailer. Classes Automobile and Truck inherit from Class Vehicle and this is represented by arrows drawn from Automobile and Truck to Vehicle. Classes Pickup and Tractor Trailer inherit from Class Truck and this is represented by arrows drawn from Pickup and Tractor Trailer to Truck.

Back to Figure

The classes shown in the diagram are labeled Object, Application, and Draw A Sprite. The Application class inherits from the Object class and it is represented by an arrow from the Application class to the Object class. The

Draw A Sprite class inherits from the Application class and it is represented by an arrow from the Draw A Sprite class to the Application class.

Back to Figure

The diagram shows 3 classes labeled Object, Bank Account, and Checking Account. The method under the Object class are as follows: to String, left parentheses, right parentheses, colon, String. The attributes under the Bank Account class are as follows: MONEY, colon, Decimal Format, equals, dollar sign, hash, comma, hash, hash, 0.00, balance, colon, double. The methods under the Bank Account class are as follows: Bank Account, left parentheses, right parentheses, Bank Account, left parentheses, double, right parentheses, get Balance, left parentheses, right parentheses, colon, double, deposit, left parentheses, double, right parentheses, colon, Bank Account, withdraw, left parentheses, double, right parentheses, colon, Bank Account, to String, left parentheses, right parentheses, colon, String. The attributes under the Checking Account class are as follows: DEFAULT underscore FEE, colon, double, equals, 5.00, monthly Fee, colon, double. The methods under the Checking Account Class are as follows: Checking Account, left parentheses, right parentheses, Checking Account, left parentheses, double, comma, double, right parentheses, apply Monthly Fee, left parentheses, right parentheses, colon, Checking Account, get Monthly Fee, left parentheses, right parentheses, colon, double, set Monthly Fee, left parentheses, double, right parentheses, colon, Checking Account, to String, left parentheses, right parentheses, colon, String.

Back to Figure

Line 1: New checking account, colon, balance is dollar sign, 0.00. Line 2: After depositing dollar sign, 350.75, colon, balance is dollar sign, 350.75. Line 3: After withdrawing dollar sign, 200.25, colon, balance is dollar sign, 150.50.

Back to Figure

Line 1: In Bank Account default constructor. Line 2: In Checking Account default constructor. Line 3: New checking account, colon, balance is dollar sign, 0.0. Line 4: blank. Line 5: In Bank Account overloaded constructor. Line 6: In Checking Account overloaded constructor. Line 7: New checking account, colon, balance is dollar sign, 100.00.

Back to Figure

Line 1: New checking account, colon. Line 2: balance is dollar sign, 100.00, semicolon, monthly fee is dollar sign, 7.50. Line 3: blank. Line 4: After charging monthly fee, colon. Line 5: balance is dollar sign, 92.50, semicolon, monthly fee is dollar sign, 7.50.

Back to Figure

Line 1: New checking account, colon. Line 2: balance is dollar sign, 100.00, semicolon, monthly fee is dollar sign, 7.50.

Back to Figure

Line 1: New checking account, colon. Line 2: balance is dollar sign, 100.00, semicolon, monthly fee is dollar sign, 7.50. Line 3: blank. Line 4: After withdrawing dollar sign, 95, colon. Line 5: balance is dollar sign, 5.00, semicolon, monthly fee is dollar sign, 7.50. Line 6: blank. Line 7: Applying the monthly fee, colon. Line 8: Warning, colon, account is overdrawn, exclamation mark. Line 9: balance is negative, dollar sign, 2.50, semicolon, monthly fee is dollar sign, 7.50.

Back to Figure

The classes shown in the diagram are as follows: Object, Bank Account, Checking Account, and Savings Account. The method under the Object class is as follows: to String, left parentheses, right parentheses, colon, String. The attributes under the Bank Account class are as follows: MONEY, colon, Decimal Format, equals, dollar sign, hash, comma, hash, hash, 0.00, balance, colon, double. The methods under the Bank Account class are as follows: Bank Account, left parentheses, right parentheses, Bank Account, left parentheses, double, right parentheses, get Balance, left parentheses, right parentheses, colon, double, deposit, left parentheses, double, right parentheses, colon, Bank Account, withdraw, left parentheses, double, right parentheses, colon, Bank Account, to String, left parentheses, right parentheses, colon, String. The Bank Account class inherits from the Object class and it is represented by an arrow from the Bank Account class to the Object class. The attributes under the Checking Account class are as follows: DEFAULT underscore FEE, colon, double, equals, 5.00, monthly Fee, colon, double. The methods under the Checking Account class are as follows: Checking Account, left parentheses, right parentheses, Checking Account, left parentheses, double, comma, double, right parentheses, apply Monthly Fee, left parentheses, right parentheses, colon, Checking Account, get Monthly Fee, left parentheses, right parentheses, colon, double, set Monthly Fee, left parentheses, double, right parentheses, colon, Checking Account, to String, left parentheses, right parentheses, colon, String. The Checking Account class inherits from the Bank Account class and it is represented by an arrow from the Checking Account Class to the Bank Account class. The attributes of the Savings Account class are as follows: DEFAULT underscore RATE, colon, double, equals, 0.03, interest Rate, colon, double. The methods under the Savings Account are as follows: Savings Account, left parentheses, right parentheses, Savings Account, left parentheses, double, comma, double, right parentheses, apply Interest, left parentheses, right parentheses, colon, Savings Account, get Interest Rate, left parentheses, right parentheses, colon, double, set Interest Rate, left parentheses, double, right parentheses, colon, Savings Account, to String, left parentheses, right parentheses, colon, String. The Savings Account class inherits from the Bank Account class and it is represented by an arrow from the Savings Account class to the Bank Account class.

Back to Figure

The window has 6 buttons which are as follows: Change interest rate, Apply interest, Deposit, Withdraw, Display account information, Exit. The table below the buttons has 3 columns labeled, Activity, Balance, and Interest Rate. The row entry is as follows: Activity: Account opened. Balance: 300.00 dollars. Interest Rate: 4.00 percent.

Back to Figure

The window has 6 buttons which are as follows: Change interest rate, Apply interest, Deposit, Withdraw, Display account information, Exit. The table below the buttons has 3 columns labeled, Activity, Balance, and Interest Rate. The row entries are as follows. Row 1. Activity: Account opened. Balance: 300.00 dollars. Interest Rate: 4.00 percent. Row 2. Activity: Deposit. Balance: 550.00 dollars. Interest Rate: 4.00 percent. Row 3. Activity: Interest applied. Balance: 551.83 dollars. Interest Rate: 4.00 percent. Row 4. Activity: Withdrawl. Balance: 451.83 dollars. Interest Rate: 4.00 percent. Row 5. Activity: Rate changed. Balance: 451.83 dollars. Interest Rate: 5.00

percent. Row 6. Activity: Interest applied. Balance: 453.72 dollars. Interest Rate: 5.00 percent.

Back to Figure

The diagram has the following classes: Object, Figure, Circle, and Square. The object class does not have attributes or methods. The Figure class inherits from the Object class and it is represented by an arrow from the Figure class to the Object class. The attributes of the Figure class are as follows: x, colon, int, y, colon, int, color, colon, color. The methods of the Figure class are as follows: Figure, left parentheses, right parentheses, Figure, left parentheses, int, comma, int, comma, Color, right parentheses, get Color, left parentheses, right parentheses, colon, Color, set Color, left parentheses, Color, right parentheses, colon, Figure, get X, left parentheses, right parentheses, colon, int, set X, left parentheses, int, right parentheses, colon, Figure, get Y, left parentheses, right parentheses, colon, int, set Y, left parentheses, int, right parentheses, colon, Figure, draw, left parentheses, Graphics Context, right parentheses, colon, void. The Circle and the Square classes inherit from the Figure class and this is represented by arrows from the Circle and the Square classes to the Figure class. The attributes of the Circle class are as follows: MIN underscore RADIUS, colon, int, radius, colon, int. The methods under the Circle class are as follows: Circle, left parentheses, right parentheses, Circle, left parentheses, int, comma, int, comma, Color, comma, int, right parentheses, get Radius, left parentheses, right parentheses, colon, int, set Radius, left parentheses, int, right parentheses, colon, Circle, draw, left parentheses, Graphics Context, right parentheses, colon, void. The attributes under the Square class are as follows: MIN underscore LENGTH, colon, int, length, colon, int. The methods under the Square class are as follows: Square, left parentheses, right parentheses, Square, left parentheses, int, comma, int, comma, Color, comma, int, right parentheses, get Length, left parentheses, right parentheses, colon, int, set Length, left parentheses, int, right parentheses, colon, Square, draw, left parentheses, Graphics Context, right parentheses, colon, void.

Back to Figure

The window has a text label, a text box, and 2 buttons. The text label reads: Enter t, left parentheses, Tortoise, right parentheses, or h, left parentheses, Hare, right parentheses. The entry in the text box reads: h. The labels of the 2 deactivated buttons are Enter Racer and Start. The image below displays a hare near a tortoise.

Back to Figure

The classes in the U M L diagram are as follows: Object, Racer, Tortoise, and Hare. The Object class does not have attributes or methods. The attributes of the Racer class are as follows: I D, colon, String, x, colon, int, y, colon, int. The methods under the Racer class are as

follows: Racer, left parentheses, right parentheses, Racer, left parentheses, String, comma, int, comma, int, right parentheses, get I D, left parentheses, right parentheses, colon, String, get X, left parentheses, right parentheses, colon, int, set X, left parentheses, int, right parentheses, colon, Racer, get Y, left parentheses, right parentheses, colon, int, set Y, left parentheses, int, right parentheses, colon, Racer, move, left parentheses, right parentheses, colon, void, draw, left parentheses, Graphics Context, right parentheses, colon, void. The Racer class inherits from the Object class and it is represented by an arrow from the Racer to the Object class. The Tortoise and Hare classes inherit from the Racer Class. The attributes under the Tortoise class are as follows: rand, colon, Random, speed, colon, int. The methods under the Tortoise class are as follows: Tortoise, left parentheses, right parentheses, Tortoise, left parentheses, String, comma, int, comma, int, right parentheses, move, left parentheses, right parentheses, colon, void, draw, left parentheses, Graphics Context, right parentheses, colon, void. The Hare class does not have attributes. The methods under the Hare class are as follows: Hare, left parentheses, right parentheses, Hare, left parentheses, String, comma, int, comma, int, right parentheses, move, left parentheses, right parentheses, colon, void, draw, left parentheses, Graphics Context, right parentheses, colon, void.

Back to Figure

The window has a text label, a text box, and 2 buttons.

The text label reads: Enter t, left parentheses, Tortoise, right parentheses, or t, left parentheses, Hare, right parentheses. The entry in the text box reads: h. The labels of the 2 deactivated buttons are Enter Racer and Start. The image below displays 3 hares and 4 tortoises.

Back to Figure

The interfaces are labeled String Handler and Validator. The classes are labeled Password Security Handler and Hex String Handler. The 4 methods under the String Handler interface are as follows. process Digit, left parentheses, char, right parentheses, colon, void. process Letter, left parentheses, char, right parentheses, colon, void. process Other, left parentheses, char, right parentheses, colon, void. parse, left parentheses, String, right parentheses, colon, void. The method under the Validator interface is as follows. is Valid, left parentheses, right parentheses, colon, boolean. The attributes under the Password Security Handler class are as follows. length, colon, int. digit, colon, boolean. other Character, colon, boolean. The methods under the Password Security Handler class are as follows. Password Security Handler, left parentheses, right parentheses. process Digit, left parentheses, char, right parentheses, colon, void. process Letter, left parentheses, char, right parentheses, colon, void. process Other, left parentheses, char, right parentheses, colon, void. security Level, left parentheses, right parentheses, colon, String. The attributes under the Hex String Handler class are as follows. valid Hex, colon, boolean, number, colon, int. The methods under the Hex String Handler class are as follows. Hex String Handler, left parentheses, right parentheses. process Digit, left parentheses, char, right parentheses, colon, void. process Letter, left parentheses, char, right parentheses, colon, void. process Other, left parentheses, char, right parentheses, colon, void. is Valid, left parentheses, right parentheses, colon, boolean. get Number, left parentheses, right parentheses, colon, int. 2 dashed arrows are drawn from the Password Security Handler class and the Hex String Handler class to the String Handler interface. A dashed arrow is drawn from the Hex String Handler class to the Validator interface.

Back to Figure

Line 1: A strong password has at least 8. Line 2: characters and contains at least one digit. Line 3. and one special character. Line 4. Enter a password, right angled bracket, open, at symbol, j b c. Line 5. open, at symbol, j b c, single quote, s, security is medium.

Back to Figure

Line 1: Enter a hexadecimal number, right angled bracket, A 5 6 E. Line 2: A 5 6 E equals 42350.

Back to Figure

The interface is labeled Shipping Cost. The 3 classes are labeled Flat Rate Shipping, Standard Shipping and

Express Shipping. The method under the Shipping Cost interface is as follows. calculate Shipping, left parentheses, double, right parentheses, colon, double. The attribute under the Flat Rate Shipping class is RATE underscore PER underscore POUND, colon, double. The method under the Flat Rate Shipping class is calculate Shipping, left parentheses, double, right parentheses, colon, double. The attributes under the Standard Shipping class are as follows. BASE underscore WEIGHT, colon, double, BASE underscore CHARGE, colon, double, EXTRA underscore POUND underscore RATE, colon, double. The method under the Standard Shipping class is calculate Shipping, left parentheses, double, right parentheses, colon, double. 2 dashed arrows are drawn from the Flat Rate Shipping and the Standard Shipping classes to the Shipping Cost interface. The attribute under the Express Shipping class is EXPRESS underscore SURCHARGE, colon, double. The method under the Express Shipping class is calculate Shipping, left parentheses, double, right parentheses, colon, double.

Back to Figure

Line 1: What is the weight of the item, question mark, 6.5. Line 2: Our shipping options are, colon. Line 3, indented once: 1 Express shipping, colon, 1 hyphen 2 business days. Line 4, indented once: 2 Standard shipping, colon, 3 hyphen 5 business days. Line 5, indented once: 3 Flat Rate, colon, 6 hyphen 8 business days. Line 6: Select your shipping option, right angled bracket, 1. Line 7: your shipping cost is dollar sign, 14.00.

Back to Figure

The classes in the U M L diagram are as follows: Object, Throwable, Exception, Runtime Exception, I O Exception, Illegal State Exception, No Such Element Exception, Arithmetic Exception, Illegal Argument Exception, Index Out Of Bounds Exception, Null Pointer Exception, File Not Found Exception, Input Mismatch Exception, Number Format Exception, String Index Out Of Bounds Exception, Array Index Out Of Bounds Exception. The Throwable class inherits from the Object class. The Exception class inherits from the Throwable class. The Runtime Exception and I O Exception classes inherit from the Exception class. The File Not Found Exception class inherits from the I O Exception class. The Illegal State Exception, No Such Element Exception, Arithmetic Exception, Illegal Argument Exception, Index Out Of Bounds Exception and Null Pointer Exception classes inherit from the Runtime Exception class. The Input Mismatch Exception class inherits from the No Such Element Exception class. The Number Format Exception class inherits from the Illegal Argument Exception class. The String Out Of Bounds Exception and Array Index Out Of Bounds Exception classes inherit from the Index Out Of Bounds Exception class.

Back to Figure

Line 1: Message, colon, data dot t x t, left parentheses, The system cannot find the file specified, right parentheses. Line 2: blank. Line 3: to String, left parentheses, right parentheses, colon, java dot i o dot File Not Found Exception, colon, data dot t x t, left parentheses, The system cannot. Line 4: find the file specified, right parentheses. Line 5: blank. Line 6: java dot i o dot File Not Found Exception, colon, data dot t x t, left parentheses, The system cannot find the. Line 7: file specified, right parentheses. Line 8, indented once: at java dot base, forward slash, java dot i o dot File Input Stream dot open 0, left parentheses, Native Method, right parentheses. Line 9, indented once: at java dot base, forward slash, java dot i o dot File Input Stream dot open, left parentheses, File Input Stream dot java, colon, 216, right parentheses. Line 10, indented once: at java dot base, forward slash, java dot i o dot File Input Stream, dot, left angled bracket, init, right angled bracket, left parentheses, File Input Stream dot java, colon, 157, right parentheses. Line 11, indented once: at java dot base, forward slash, java dot util dot Scanner dot, left angled bracket, init, right angled bracket, left parentheses, Scanner dot java, colon, 639, right parentheses. Line 12, indented once: at Echo File Data dot main, left parentheses, Echo File Data dot java, colon, 16, right parentheses.

Back to Figure

Line 1: The Matrix. Line 2: 136. Line 3: Finding Nemo.

Line 4: 100. Line 5: Titanic. Line 6: 194. Line 7: Casablanca. Line 8: 102. Line 9: Gone With the Wind. Line 10: 220.

Back to Figure

Line 1: The Matrix, comma, 136 minutes. Line 2: Finding Nemo, comma, 100 minutes. Line 3: Titanic, comma, 194 minutes. Line 4: Casablanca, comma, 102 minutes. Line 5: Gone With the Wind, comma, 220 minutes.

Back to Figure

Unable to find movies dot t x t, comma, exiting.

Back to Figure

The classes in the U M L diagram are labeled Object, Scanner, Input Stream, File Input Stream and Object Input Stream. The File Input Stream and Object Input Stream classes inherit from the Input Stream Class. The Scanner class and Input Stream class inherit from the Object class. The packages associated with the Scanner and the Input Stream classes are java dot util and java dot i o.

Back to Figure

The classes in the U M L diagram are labeled Object, Writer, Output Stream, Print Writer, Output Stream Writer, File Output Stream, Object Output Stream, and File Writer. The File Writer class inherits from the Output

Stream Writer class. The Print Writer and Output Stream Writer classes inherit from the Writer class. The File Output Stream and the Object Output Stream classes inherit from the Output Stream class. The Writer and the Output Stream classes inherit from the Object class.

Back to Figure

Line 1: 300. Line 2: 15000, blank space, 1 point 0, blank space, 0 point 0, blank, 0 point 0. Line 3: 15000, blank space, 1 point 0, blank space, 1 point 0, blank space, 1 point 0. Line 4: 15000, blank space, 0 point 0, blank space, 1 point 0, blank space, 0 point 0.

Back to Figure

The background color is blue. The colors of the flag are red, white, and green.

Back to Figure

Line 1: 30. Line 2: 10 0 point 0 1 point 0 0 point 0. Line 3: 10 1 point 0 1 point 0 1 point 0. Line 4: 10 1 point 0 0 point 0 0 point 0. Line 5: 10 0 point 0 1 point 0 0 point 0. Line 6: 10 1 point 0 1 point 0 1 point 0. Line 7: 10 1 point 0 0 point 0 0 point 0. Line 8: 10 0 point 0 1 point 0 0 point 0. Line 9: 10 1 point 0 1 point 0 1 point 0. Line 10: 10 1 point 0 0 point 0 0 point 0. Line 11: 10 0 point 0 1 point 0 0 point 0. Line 12: 10 1 point 0 1 point 0 1 point 0. Line 13: 10 1 point 0 0 point 0 0 point 0. Line 14: 10 0 point 0 1 point 0 0 point 0. Line 15: 10 1 point 0 1 point 0 1 point 0.

Line 16: 10 1 point 0 0 point 0 0 point 0. Line 17: 10 0 point 0 1 point 0 0 point 0. Line 18: 10 1 point 0 1 point 0 1 point 0. Line 19: 10 1 point 0 0 point 0 0 point 0. Line 20: 10 0 point 0 1 point 0 0 point 0. Line 21: 10 1 point 0 1 point 0 1 point 0. Line 22: 10 1 point 0 0 point 0 0 point 0. Line 23: 10 0 point 0 1 point 0 0 point 0. Line 24: 10 1 point 0 1 point 0 1 point 0. Line 25: 10 1 point 0 0 point 0 0 point 0. Line 26: 10 0 point 0 1 point 0 0 point 0. Line 27: 10 1 point 0 1 point 0 1 point 0. Line 28: 10 1 point 0 0 point 0 0 point 0. Line 29: 10 0 point 0 1 point 0 0 point 0. Line 30: 10 1 point 0 1 point 0 1 point 0. Line 31: 10 1 point 0 0 point 0 0 point 0.

Back to Figure

Line 1: A A 123. Line 2: B W I. Line 3: S F O. Line 4: 235. Line 5: 239.5.

Back to Figure

Line 1: A A 123, comma, B W I, comma, S F O, comma, 235, comma, 239.5. Line 2: A A 200, comma, B O S, comma, J F K, comma, 150, comma, 89.3. Line 3: A A 900, comma, L A X, comma, C H I, comma, 201, comma, 201.8.

Back to Figure

Line 1: Flight, left square bracket, flight Number, equals, A A 123, comma, origin, equals, B W I, comma, destination, equals, S F O, comma. Line 2, indented once: num

Passengers, equals, 235, comma, a v g Ticket Price, equals, 239.5, right square bracket. Line 3: Flight, left square bracket, flight Number, equals, A A 200, comma, origin, equals, B O S, comma, destination, equals, J F K, comma. Line 4, indented once: num Passengers, equals, 150, comma, a v g Ticket Price, equals, 89.3, right square bracket. Line 5: Flight, left square bracket, flight Number, equals, A A 900, comma, origin, equals, L A X, comma, destination, equals, C H I, comma. Line 6, indented once: num Passengers, equals, 201, comma, a v g Ticket Price, equals, 201.8, right square bracket.

Back to Figure

The captions within the window are as follows: Check, colon, Amount, equals, negative 200.00 dollars. Your current account balance, equals, negative 200.00 dollars. Check. A rectangular box below is marked A B C Bank.

Back to Figure

Line 1: check, hash symbol, 13, colon, negative 200.00. Line 2: Check, hash symbol, 14, colon, negative 100.00. Line 3: Withdrawal June 12, colon, negative 200.00. Line 4: Withdrawal June 17, colon, negative 400.00. Line 5: Withdrawal June 23, colon, negative 100. Line 6: Deposit, colon, 4000.00. Line 7: Deposit, colon, 100.00. Line 8: Something else, colon, negative 1000.00. Line 9: Check, hash symbol, 16, colon, negative 500.00. Line 10: Check, hash symbol, 15, colon, negative 100.00.

Back to Figure

Line 1: Paris. Line 2: Baltimore. Line 3: Seattle. Line 4: San Francisco. Line 5: Boston. Line 6: Los Angeles. Line 7: Madrid. Line 8: Barcelona. Line 9: Buenos Aires.

Back to Figure

Line 1: 1100.50. Line 2: negative 100.48. Line 3: 200.39. Line 4: 150.00.

Back to Figure

Line 1: Number of cities starting with B, colon, 4. Line 2: Total expenses are, dollar sign, 1,450.89.

Back to Figure

Line 1: city 1 is New York. Line 2: state is F L.

Back to Figure

Line 1: Left square bracket, Left brace, double quotes, title, double quotes, colon, double quotes After Dark in Central Park, double quotes, comma, double quotes, year, double quotes, colon, 1900, comma, double quotes, cast, double quotes, colon, Left square bracket, Right square bracket comma. Line 2: double quotes, genres, double quotes, colon, Left square bracket, Right square bracket, right brace, comma, Left brace, double quotes, title, double quotes, colon, double quotes, Boarding School Girls Pajama Parade, double quotes, comma. Line

3: double quotes, year, double quotes, colon, 1900, comma, double quotes, cast, double quotes, colon, Left square bracket, Right square bracket, comma, double quotes, genres, double quotes, colon, Left square bracket, Right square bracket, right brace, comma, Left brace, double quotes, title, double quotes, colon, double quotes Buffalo Bill's. Line 4: Wild West Parade, double quotes, comma, double quotes, year, double quotes, colon, 1900, comma, double quotes, cast, double quotes, colon, Left square bracket, Right square bracket, comma, double quotes, genres, double quotes, colon, Left square bracket, Right square bracket, right brace, comma. Line 5: dot, dot, dot. Line 6: Left brace, double quotes, title, double quotes, double quotes, Holmes and Watson, double quotes, comma, double quotes, year, double quotes, colon, 2018, comma, double quotes, cast, double quotes, colon, Left square bracket, double quotes, Will. Line 7: Ferrell, double quotes, comma, double quotes, John C. Reilly, double quotes, comma, double quotes, Rebecca Hall, double quotes, comma, double quotes, Ralph Fiennes, double quotes, comma, double quotes, Rob. Line 8: Brydon, double quotes, comma, double quotes, Kelly Macdonald, double quotes, comma, double quotes, Lauren Lapkus, double quotes comma, double quotes Hugh Laurie, double quotes, Right square bracket comma. Line 9: double quotes, genres, double quotes, colon, Left square bracket, double quotes, Action, double quotes, comma, double quotes, Mystery, double quotes, comma, double quotes, Comedy, double quotes, Right square bracket, right brace, comma, Left brace, double quotes, title, double quotes, colon, double quotes, On the. Line 10: Basis of Sex, double quotes, comma, double quotes, year, double quotes, colon, 2018, comma, double quotes, cast, double quotes, colon, Left square bracket, double quotes, Felicity Jones, double quotes, comma, double quotes, Armie. Line 11: Hammer, double quotes, comma, double quotes, Justin Theroux, double quotes, comma, double quotes, Jack Reynor, double quotes, comma, double quotes, Cailee Spaeny, double quotes, comma. Line 12: double quotes, Sam Waterston, double quotes, comma, double quotes, Kathy Bates, double quotes, Right square bracket, comma, double quotes, genres, double quotes, colon, Left square bracket, double quotes, Biography, double quotes, comma. Line 13: double quotes, Drama, double quotes, Right square bracket, right brace, comma, Left brace, double quotes, title, double quotes, colon, double quotes, Destroyer, double quotes, comma, double quotes, year, double quotes, colon, 2018, comma, double quotes, cast, double quotes, colon, Left square bracket, double quotes, Nicole. Line 14: Kidman, double quotes, comma, double quotes, Tatiana Maslany, double quotes, comma, double quotes, Sebastian Stan, double quotes, comma, double quotes, Toby Kebbell, double quotes, comma. Line 15: double quotes, Scoot Mc Nairy, double quotes, Right square bracket, comma, double quotes, genres, double quotes, colon, Left square bracket, double quotes, Crime, double quotes, comma, double quotes, Thriller, double quotes, Right square bracket, right brace, Right square bracket.

Back to Figure

Line 1: There are 28795 movies. Line 2: Please enter a year, right angled bracket, 2000. Line 3: There were 213 movies in 2000.

Back to Figure

Line 1: End of file reached. Line 2: Closing file. Line 3: There are 2 flights from B W I. Line 4: Average number of passengers, colon, 200.0.

Back to Figure

The captions within the window are as follows: Check, colon, Amount, equals, negative 500.00 dollars. Your current account balance, equals, negative 500.00 dollars. Check. A rectangular box below is marked A B C Bank.

Back to Figure

The first sample run is as follows. Line 1: Enter your e mail address, right angled bracket, mary dot j b dot com. Line 2: Email address does not contain, at symbol. The second sample run is as follows. Line 1: Enter your e mail address, right angled bracket, john, at symbol, j b dot com. Line 2: Your host is j b dot com.

Back to Figure

Line 1: Hello World. Line 2: Hello World. Line 3: Hello World. Line 4: Hello World. Line 5: Hello World.

Back to Figure

Step 1: A call to print, double quotes, Hello World, double quotes, 5 times. Step 2: Print double quotes, Hello World, double quotes, once. A call to print, double quotes, Hello World, double quotes, 4 times. Step 3: Print double quotes, Hello World, double quotes, once. A call to print, double quotes, Hello World, double quotes, 3 times. Step 4: Print, double quotes, Hello World, double quotes, once. A call to print, double quotes, Hello World, double quotes, 2 times. Step 5: Print, double quotes, Hello World, double quotes, once. A call to print, double quotes, Hello World, double quotes, 1 time. Step 6: print, double quotes, Hello World, double quotes, once. A call to print, double quotes, Hello World, double quotes, 0 times. Step 7: return.

Back to Figure

Factorial, left parentheses, 5, right parentheses, is 120.

Back to Figure

Line 1: factorial, left parentheses, 5, right parentheses, called. Line 2: Current Stack. Line 3: 14 main. Line 4: factorial, left parentheses, 4, right parentheses, called. Line 5: Current Stack. Line 6: 46 factorial. Line 7: 14 main. Line 8: factorial, left parentheses, 3, right parentheses, called. Line 9: Current Stack. Line 10: 46

factorial. Line 11: 46 factorial. Line 12: 14 main. Line 13: factorial, left parentheses, 2, right parentheses, called. Line 14: Current Stack. Line 15: 46 factorial. Line 16: 46 factorial. Line 17: 46 factorial. Line 18: 14 main. Line 19: factorial, left parentheses, 1, right parentheses, called. Line 20: Current Stack. Line 21: 46 factorial. Line 22: 46 factorial. Line 23: 46 factorial. Line 24: 46 factorial. Line 25: 14 main. Line 26: factorial, left parentheses, 0, right parentheses, called. Line 27: Current Stack. Line 28: 46 factorial. Line 29: 46 factorial. Line 30: 46 factorial. Line 31: 46 factorial. Line 32: 46 factorial. Line 33: 14 main. Line 34: Base case detected. Line 35: factorial, left parentheses, 0, right parentheses, returning 1. Line 36: Current Stack. Line 37: 46 factorial. Line 38: 46 factorial. Line 39: 46 factorial. Line 40: 46 factorial. Line 41: 46 factorial. Line 42: 14 main. Line 43: factorial, left parentheses, 1, right parentheses, returning 1 asterisk 1. Line 44: Current Stack. Line 45: 46 factorial. Line 46: 46 factorial. Line 47: 46 factorial. Line 48: 46 factorial. Line 49: 14 main. Line 50: factorial, left parentheses, 2, right parentheses, returning 2 asterisk 1. Line 51: Current Stack. Line 52: 46 factorial. Line 53: 46 factorial. Line 54: 46 factorial. Line 55: 14 main. Line 56: factorial, left parentheses, 3, right parentheses, returning 3 asterisk 2. Line 57: Current Stack. Line 58: 46 factorial. Line 59: 46 factorial. Line 60: 14 main. Line 61: factorial, left parentheses, 4, right parentheses, returning 4 asterisk 6. Line 62: Current Stack. Line 63: 46 factorial. Line 64: 14 main. Line 65: factorial, left parentheses, 5, right parentheses, returning 5 asterisk 24. Line 66: Current Stack. Line 67: 14 main. Line 68: Factorial, left parentheses, 5, right parentheses, is 120.

Back to Figure

Line 1: Exception in thread, double quotes, main, double quotes, java dot lang dot Stack Overflow Error. Line 2, indented once: at Recursive Factorial dot factorial, left parentheses, Recursive Factorial dot java, colon, 23, right parentheses. Line 3, indented once: at Recursive Factorial dot factorial, left parentheses, Recursive Factorial dot java, colon, 23, right parentheses. Line 4, indented once: at Recursive Factorial dot factorial, left parentheses, Recursive Factorial dot java, colon, 23, right parentheses. Line 5, indented once: at Recursive Factorial dot factorial, left parentheses, Recursive Factorial dot java, colon, 23, right parentheses.

Back to Figure

The G C D of 474 and 162 is 6.

Back to Figure

Line 1: g c d, left parentheses, 474, comma, 162 , right parentheses, 474 percentage sign 162 equals 150. Line 2: Current stack. Line 3: 13 main. Line 4: g c d, left parentheses, 162, comma, 150 , right parentheses, 162 percentage sign 150 equals 12. Line 5: Current stack. Line 6: 48 g c d. Line 7: 13 main. Line 8: g c d, left parentheses, 150, comma, 12 , right parentheses, 150 percentage sign 12 equals 6. Line 9: Current stack. Line 10: 48 g c d. Line 11: 48 g c d. Line 12: 13 main. Line 13: g c d, left parentheses, 12, comma, 6 , right parentheses, 12 percentage sign 6 equals 0. Line 14: Current stack. Line 15: 48 g c d. Line 16: 48 g c d. Line 17: 48 g c d. Line 18: 13 main. Line 19: base case reached, comma, returning 6. Line 20: Current stack. Line 21: 48 g c d. Line 22: 48 g c d. Line 23: 48 g c d. Line 24: 13 main. Line 25: g c d, left parentheses, 12, comma,6, right parentheses, returning 6. Line 26: Current stack. Line 27: 48 g c d. Line 28: 48 g c d. Line 29: 13 main. Line 30: g c d, left parentheses, 150, comma,12, right parentheses, returning 6. Line 31: Current stack. Line 32: 48 g c d. Line 33: 13 main. Line 34: g c d, left parentheses, 162, comma,150, right parentheses, returning 6. Line 35: Current stack. Line 36: 13 main. Line 37: The G C D of 474 and 162 is 6.

Back to Figure

C, left parentheses, 5, comma, 2, right parentheses, equals, 10.

Back to Figure

The window has a text label, a text box and a button. The text label is as follows: Enter a word. The button is labeled check.

Back to Figure

The window has a text label, a text box and a button. The text label is as follows: Enter a word. The entry in the text box reads: able was i ere i saw elba. The button is labeled check. The following captions are arranged in an inverted triangular pattern. Line 1: able was i ere i saw elba. Character a at the beginning and end is colored red. Line 2: b l e was i ere i saw e l b. Character b at the beginning and end is colored red. Line 3: l e was i ere i saw e l. Character l at the beginning and end is colored red. Line 4: e was i ere i saw e. Character e at the beginning and end is colored red. Line 5: was i ere i saw. Line 6: was i ere i saw. Character w at the beginning and end is colored red. Line 7: as i ere i s a. Character a at the beginning and end is colored red. Line 8: s i ere i s. Character s at the beginning and end is colored red. Line 9: i ere i. Line 10: i ere i. Character i at the beginning and end is colored red. Line 11: ere. Line 12: ere. Character e at the beginning and end is colored red. Line 13: r. Character r is colored red. The captions at the bottom of the screen are as follows: Your result, double quotes, able was i ere i saw elba, double quotes, is a palindrome. The correct result, colon, double quotes, able was i ere i saw elba, double quotes, is a palindrome.

Back to Figure

The window has a text label, a text box and a button. The text label is as follows: Enter a word. The entry in the text box reads: able was i here i saw elba. The button is labeled check. The following captions are arranged in an inverted triangular pattern. Line 1: able was i here i saw elba. Line 2: b l e was i here i saw e l b. Line 3: l e was i here i saw e l. Line 4: e was i here i saw e. Line 5: was i here i saw. Line 6: was i here i saw. Line 7: as i here i s a. Line 8: s i here i s. Line 9: i here i. Line 10: i here i. Line 11: here. Line 12: here. The captions at the bottom of the screen are as follows: Your result, double quotes, able was i here i saw elba, double quotes, is not a palindrome. The correct result, colon, double quotes, able was i here i saw elba, double quotes, is not a palindrome.

Back to Figure

Line 1: sum called with index 0. Line 2: sum called with index 1. Line 3: sum called with index 2. Line 4: sum called with index 3. Line 5: sum called with index 4. Line 6: total, equals, 35. Lines 1 and 5 are highlighted yellow.

Back to Figure

The output produced by the first run is as follows. Line 1: Enter a value to search for, right angled bracket, 7. Line 2: 7 found at index 2. The output produced by the second run is as follows. Line 1: Enter a value to search for, right angled bracket, 34. Line 2: 34 not found.

Back to Figure

The window has a text label, a text box, and a button. The text label reads: Enter an integer between 1 and 9. The entry in the text box is 5. The button is labeled run and it is inactive. The image in the window shows 3 vertical sticks labeled 0, 1, and 2. Stick 0 has rings labeled 1, 2, 3, 4, and 5, from top to bottom.

Back to Figure

The window has a text label, a text box, and a button. The text label reads: Enter an integer between 1 and 9. The entry in the text box is 5. The button is labeled run and it is inactive. The image in the window shows 3 vertical sticks labeled 0, 1, and 2. Stick 0 has a ring labeled 5. Stick 1 has rings labeled 1, 2, 3 and 4 from top to bottom.

Back to Figure

The window has a text label, a text box, and a button. The text label reads: Enter an integer between 1 and 9. The entry in the text box is 5. The button is labeled run and it is inactive. The image in the window shows 3 vertical sticks labeled 0, 1, and 2. Stick 1 has rings labeled 1, 2, 3 and 4 from top to bottom. Stick 2 has a ring labeled 5.

Back to Figure

The window has a text label, a text box, and a button. The text label reads: Enter an integer between 1 and 9. The entry in the text box is 5. The button is labeled run. The image in the window shows 3 vertical sticks labeled 0, 1, and 2. Stick 2 has 5 rings labeled 1, 2, 3, 4, and 5 from top to bottom.

Back to Figure

The window has a text label, a text box and a button. The text label is as follows: Enter a number of disks. The button is labeled run.

Back to Figure

The window has a text label, a text box, and a button. The text label reads: Enter an integer between 1 and 9. The entry in the text box is 5. The button is labeled run and it is inactive. The image in the window shows 3 vertical sticks labeled 0, 1, and 2. Stick 0 has a ring labeled 1. Stick 1 has rings labeled 3 and 4 from top to bottom. Stick 2 has rings labeled 2 and 5 from top to bottom.

Back to Figure

Line 1: 10, 15, 23, 54, 66, 75, 80, 98.

Back to Figure

“Line 1: merge Sort, left parentheses, 0, comma, 7 , right parentheses, called. Line 2: merge Sort, left parentheses, 0, comma, 3 , right parentheses, called. Line 3: merge Sort, left parentheses, 0, comma, 1 , right parentheses, called. Line 4: merge Sort, left parentheses, 0, comma, 0 , right parentheses, called. Line 5: merge Sort, left parentheses, 1, comma, 1 , right parentheses, called. Line 6: merge, left parentheses, 0, comma, 0, comma, 1, comma, 1 , right parentheses, called. Line 7: merge Sort, left parentheses, 2, comma, 3 , right parentheses, called. Line 8: merge Sort, left parentheses, 2, comma, 2 , right parentheses, called. Line 9: merge Sort, left parentheses, 3, comma, 3 , right parentheses, called. Line 10: merge, left parentheses, 2, comma, 2, comma, 3, comma, 3 , right parentheses, called. Line 11: merge, left parentheses, 0, comma, 1, comma, 2, comma, 3 , right parentheses, called. Line 12: merge Sort, left parentheses, 4, comma, 7 , right parentheses, called. Line 13: merge Sort, left parentheses, 4, comma, 5 , right parentheses, called. Line 14: merge Sort, left parentheses, 4, comma, 4 , right parentheses, called. Line 15: merge Sort, left parentheses, 5, comma, 5 , right parentheses, called. Line 16: merge, left parentheses, 4, comma, 4, comma, 5, comma, 5 , right parentheses, called. Line 17: merge Sort, left parentheses, 6, comma, 7 , right parentheses, called. Line 18: merge Sort, left parentheses, 6, comma, 6 , right parentheses, called. Line 19: merge Sort, left parentheses, 7, comma, 7 , right parentheses, called. Line 20: merge, left parentheses, 6, comma, 6, comma, 7, comma, 7 , right parentheses, called. Line 21: merge, left parentheses, 4, comma, 5, comma, 6, comma, 7 , right parentheses, called. Line 22: merge, left parentheses, 0, comma, 3, comma, 4, comma, 7 , right parentheses, called. Line 23: 10, 15, 23, 54, 66, 75, 80, 98.”

Back to Figure

Line 1: 10, 15, 23, 54, 66, 75, 80, 98.

Back to Figure

“Line 1: quick Sort, left parentheses, 0, comma, 7, right parentheses, called. Line 2: partition, left parentheses, 0, comma, 7, right parentheses, called. Line 3: quick Sort , left parentheses, 0, comma, 3, right parentheses, called. Line 4: partition, left parentheses, 0, comma, 3, right parentheses, called. Line 5: quick Sort , left parentheses, 0, comma, negative 1, right parentheses, called. Line 6: quick Sort , left parentheses, 1, comma, 3, right parentheses, called. Line 7: partition, left parentheses, 1, comma, 3, right parentheses, called. Line 8: quick Sort , left parentheses, 1, comma, 2, right parentheses, called. Line 9: partition, left parentheses, 1, comma, 2, right parentheses, called. Line 10: quick Sort , left parentheses, 1, comma, 0, right parentheses, called. Line 11: quick Sort , left parentheses, 2, comma, 2, right parentheses, called. Line 12: quick Sort , left parentheses, 4, comma, 3, right parentheses, called. Line 13: quick Sort , left parentheses, 5, comma, 7, right parentheses, called. Line 14: partition, left parentheses, 5, comma, 7, right parentheses, called. Line 15: quick Sort , left parentheses, 5, comma, 4, right parentheses, called. Line 16: quick Sort , left parentheses, 6, comma, 7, right parentheses, called. Line 17: partition, left parentheses, 6, comma, 7, right parentheses, called. Line 18: quick Sort , left parentheses, 6, comma, 6, right parentheses, called. Line 19: quick Sort , left parentheses, 8, comma, 7, right parentheses, called. Line 20: 10, 15, 23, 54, 66, 75, 80, 98.”

Back to Figure

A node is represented by a rectangle which is partitioned into 2 parts. The rectangle is marked 5. An arrow points away from the node.

Back to Figure

Each node is represented by a rectangle which is partitioned into a larger and a smaller rectangle. The list has 4 nodes. The marking within Node 1 is 7, Sarah, Mario. The marking within Node 2 is 5, Ajay, Sonic. The marking within Node 3 is 8, Gino, Diablo. The marking within Node 4 is 2, Jin, Golf. An arrow points from Node 1 to Node 2. A second arrow points from Node 2 to Node 3. A third arrow points from Node 3 to Node 4. The fifth arrow points to null.

Back to Figure

Each node is represented by a rectangle which is partitioned into a larger and a smaller rectangle. The list has 4 nodes. The marking within Node 1 is 7. The marking within Node 2 is 5. The marking within Node 3 is 8. The marking within Node 4 is 2. An arrow points from Node 1 to Node 2. A second arrow points from Node 2 to Node 3. A third arrow points from Node 3 to Node 4. An arrow labeled head points to Node 1. The fifth arrow points to null.

Back to Figure

Each node is represented by a rectangle which is partitioned into a larger and a smaller rectangle. The list has 4 nodes. The marking within Node 1 is 7. The marking within Node 2 is 5. The marking within Node 3 is 8. The marking within Node 4 is 2. An arrow points from Node 1 to Node 2. A second arrow points from Node 2 to Node 3. A third arrow points from Node 3 to Node 4. An arrow labeled head points to Node 1. The fifth arrow points to null. A fifth node is marked 6. An arrow from the fifth node point to null.

Back to Figure

Each node is represented by a rectangle which is partitioned into a larger and a smaller rectangle. The list has 5 nodes. The marking within Node 1 is 6. The marking within Node 2 is 7. The marking within Node 3 is 5. The marking within Node 4 is 8. The marking within node 5 is 2. An arrow points from Node 1 to Node 2. A second arrow points from Node 2 to Node 3. A third arrow points from Node 3 to Node 4. A fourth arrow points from Node 4 to Node 5. An arrow labeled head points to Node 2. The fifth arrow points to null.

Back to Figure

Each node is represented by a rectangle which is partitioned into a larger and a smaller rectangle. The list has 5 nodes. The marking within Node 1 is 6. The marking within Node 2 is 7. The marking within Node 3 is 5. The marking within Node 4 is 8. The marking within node 5 is 2. An arrow points from Node 1 to Node 2. A second arrow points from Node 2 to Node 3. A third arrow points from Node 3 to Node 4. A fourth arrow points from Node 4 to Node 5. An arrow labeled head points to Node 1. The fifth arrow points to null.

Back to Figure

Each node is represented by a rectangle which is partitioned into a larger and a smaller rectangle. The list has 4 nodes. The marking within Node 1 is 7. The marking within Node 2 is 5. The marking within Node 3 is 8. The marking within Node 4 is 2. An arrow labeled head points to Node 1. An arrow labeled previous points to Node 2. An arrow labeled current points to Node 3. The first arrow points from Node 1 to Node 2. The second arrow points from Node 2 to Node 3. The third arrow points from Node 3 to Node 4. An arrow labeled null points away from Node 4.

Back to Figure

Each node is represented by a rectangle which is partitioned into a larger and a smaller rectangle. The list has 4 nodes. The marking within Node 1 is 7. The marking within Node 2 is 5. The marking within Node 3 is 8. The marking within Node 4 is 2. An arrow labeled head points to Node 1. An arrow labeled previous points to

Node 2. An arrow labeled current points to Node 3. The first arrow points from Node 1 to Node 2. The second arrow points from Node 2 to Node 4. The third arrow points from Node 3 to Node 4. An arrow labeled null points away from Node 4.

Back to Figure

Each node is represented by a rectangle which is partitioned into a larger and a smaller rectangle. The list has 4 nodes. The marking within Node 1 is 7. The marking within Node 2 is 5. The marking within Node 3 is 8. The marking within Node 4 is 2. An arrow labeled head, current points to Node 1. The first arrow points from Node 1 to Node 2. The second arrow points from Node 2 to Node 3. The third arrow points from Node 3 to Node 4. An arrow labeled null points away from Node 4.

Back to Figure

Each node is represented by a rectangle which is partitioned into a larger and a smaller rectangle. The list has 4 nodes. The marking within Node 1 is 7. The marking within Node 2 is 5. The marking within Node 3 is 8. The marking within Node 4 is 2. An arrow labeled head points to Node 2. The first arrow points from Node 1 to Node 2. The second arrow points from Node 2 to Node 3. The third arrow points from Node 3 to Node 4. An arrow labeled null points away from Node 4.

Back to Figure

Line 1: Number of items in the list, colon, 0. Line 2: Number of items in the list, colon, 1. Line 3: 7. Line 4: Number of items in the list, colon, 2. Line 4: 2, blank space, 7. Line 5: Number of items in the list, colon, 3. Line 6: 5, blank space, 2, blank space, 7. Line 7: 8 could not be deleted, colon. Line 8: 2 was successfully deleted, colon. Line 9: Number of items in the list, colon, 2. Line 10: 5, blank space, 7. Line 11: 7 was successfully deleted, colon. Line 12: Number of items in the list, colon, 1. Line 13: 5. Line 14: 5 was successfully deleted, colon. Line 15: Number of items in the list, colon, 0. Line 16: 8 could not be deleted, colon. Line 17: Number of items in the list, colon, 0.

Back to Figure

Line 1: Number of items in the list, colon, 0. Line 2: blank. Line 3: Number of items in the list, colon, 1. Line 3: i d, colon, 7. Tab space. name, colon, Sarah. Tab space. game, colon, Mario. Line 4: blank. Line 5: Number of items in the list, colon, 2. Line 6: i d, colon, 2. Tab space. name, colon, Jin. Tab space. game, colon, Golf. Line 7: i d, colon, 7. Tab space. name, colon, Sarah. Tab space. game, colon, Mario. Line 8: blank. Line 9: Number of items in the list, colon, 3. Line 10: i d, colon, 5. Tab space. name, colon, Ajay. game, colon, Sonic. Line 11: i d, colon, 2. Tab space. name, colon, Jin. Tab space. game, colon, Golf. Line 12: i d, colon, 7. Tab space. name, colon, Sarah. Tab space. game, colon, Mario. Line 13: blank. Line 14: 8 not found, colon, cannot be deleted. Line 15:

blank. Line 16: Player retrieved, colon, i d, colon, 2, name, colon, Jin. Tab spaces. game, colon, Golf. Line 17: Number of items in the list, colon, 3. Line 18: i d, colon, 5. Tab spaces. name, colon, Ajay. Tab spaces. game, colon, Sonic. Line 19: i d, colon, 2. Tab spaces. name, colon, Jin. Tab space. game, colon, Golf. Line 20: i d, colon, 7. Tab spaces. name, colon, Sarah. Tab space. game, colon, Mario. Line 21: blank. Line 22: Player deleted, colon, i d, colon, 2 name, colon, Jin. Tab spaces. game, colon, Golf. Line 23: Number of items in the list, colon, 2. Line 24: i d, colon, 5. Tab spaces. name, colon, Ajay. Tab space. game, colon, Sonic. Line 25: i d, colon, 7. Tab spaces. name, colon, sarah. Tab space. game, colon, Mario. Line 26: blank. Line 27: Player deleted, colon, i d, colon, 7. Tab spaces. name, colon, Sarah. Tab space. game, colon, Mario. Line 28: Number of items in the list, colon, 1. Line 29: i d, colon, 5. Tab spaces. name, colon, Ajay. Tab space. game, colon, Sonic. Line 30: blank. Line 31: Player deleted, colon, i d, colon, 5. Tab spaces. name, colon, Ajay. Tab space. game, colon, Sonic. Line 32: Number of items in the list, colon, 0. Line 33: blank. Line 34: 7 not found, colon, cannot be deleted.

Back to Figure

5 trays are arranged one above the other. An arrow pointing to the topmost plate reads: top of the stack.

Back to Figure

A rectangular box labeled A T M is on the right. 5 stick figures are shown one after the other from left to right. The leftmost stick Figure is marked back. The rightmost stick Figure is marked front.

Back to Figure

The queue has 4 nodes. Each node is represented by a rectangle which is partitioned into 2 parts. The markings within the first node are 7, Sarah, Mario. The markings within the second node are 5, Ajay, Sonic. The markings within the third node are 8, Gino, Diablo. The markings within the fourth node are, 2, Jin, Golf. An arrow labeled head points to Node 1. An arrow labeled tail points to Node 4. An arrow labeled null points away from Node 4. An arrow points from Node 1 to Node 2. A second arrow points from Node 2 to Node 3. A third arrow points from Node 3 to Node 4.

Back to Figure

The queue has 4 nodes. Each node is represented by a rectangle which is partitioned into 2 parts. The markings within the first node are 7, Sarah, Mario. The markings within the second node are 5, Ajay, Sonic. The markings within the third node are 8, Gino, Diablo. The markings within the fourth node are, 2, Jin, Golf. An arrow labeled head points to Node 1. An arrow labeled tail points to Node 4. An arrow labeled null points away from Node 4. An arrow points from Node 1 to Node 2. A second arrow points from Node 2 to Node 3. A third arrow points from Node 3 to Node 4. A fifth node is marked 6, Steve, N F L. An arrow pointing to Node 5 is marked p n. An arrow labeled null points away from Node 5.

Back to Figure

The queue has 5 nodes. Each node is represented by a rectangle which is partitioned into 2 parts. The markings within the first node are 7, Sarah, Mario. The markings within the second node are 5, Ajay, Sonic. The markings within the third node are 8, Gino, Diablo. The markings within the fourth node are 2, Jin, Golf. The markings within the fifth node are 6, Steve, N F L. An arrow labeled head points to Node 1. An arrow labeled tail points to Node 4. An arrow labeled p n points to Node 5. An arrow labeled null points away from Node 5. An arrow points from Node 1 to Node 2. A second arrow points from Node 2 to Node 3. A third arrow points from Node 3 to Node 4. A fourth arrow points from Node 4 to Node 5.

Back to Figure

The queue has 5 nodes. Each node is represented by a rectangle which is partitioned into 2 parts. The markings within the first node are 7, Sarah, Mario. The markings within the second node are 5, Ajay, Sonic. The markings within the third node are 8, Gino, Diablo. The markings within the fourth node are 2, Jin, Golf. The markings within the fifth node are 6, Steve, N F L. An arrow labeled head points to Node 1. An arrow labeled tail points to Node 5.An arrow labeled null points away from Node 5. An arrow points from Node 1 to Node 2. A second arrow points from Node 2 to Node 3. A third arrow points from Node 3 to Node 4. A fourth arrow points from Node 4 to Node 5.

Back to Figure

The table has 3 columns. Column 1 is unmarked. Columns 2 and 3 are labeled Index and Player Object, respectively. The first 4 rows are empty. The entries of the remaining 3 rows are as follows. Row 5. Unmarked column: top. Index: 2. Player object: left parentheses, 8, comma, Gino, comma, Diablo, right parentheses. Row 5 is shaded. Row 6. Unmarked column: blank. Index: 1. Player object: left parentheses, 7, comma, Sarah, comma, Mario, right parentheses. Row 7. Unmarked column: blank. Index: 0. Player object: left parentheses, 2, comma, Jin, comma, Golf, right parentheses.

Back to Figure

The table has 3 columns. Column 1 is unmarked. Columns 2 and 3 are labeled Index and Player Object, respectively. The first 3 rows are empty. The entries of the remaining 4 rows are as follows. Row 4. Unmarked column: top. Index: 3. Player object: left parentheses, 6, comma, Steve, comma, N F L, right parentheses. Row 4 is shaded. Row 5. Unmarked column: blank. Index: 2.

Player object: left parentheses, 8, comma, Gino, comma, Diablo, right parentheses. Row 6. Unmarked column: blank. Index: 1. Player object: left parentheses, 7, comma, Sarah, comma, Mario, right parentheses. Row 7. Unmarked column: blank. Index: 0. Player object: left parentheses, 2, comma, Jin, comma, Golf, right parentheses.

Back to Figure

The table has 3 columns. Column 1 is unmarked. Columns 2 and 3 are labeled Index and Player Object, respectively. The first 3 rows are empty. The entries of the remaining 4 rows are as follows. Row 4. Unmarked column: blank. Index: 3. Player object: left parentheses, 6, comma, Steve, comma, N F L, right parentheses. Row 5. Unmarked column: top. Index: 2. Player object: left parentheses, 8, comma, Gino, comma, Diablo, right parentheses. Row 5 is shaded. Row 6. Unmarked column: blank. Index: 1. Player object: left parentheses, 7, comma, Sarah, comma, Mario, right parentheses. Row 7. Unmarked column: blank. Index: 0. Player object: left parentheses, 2, comma, Jin, comma, Golf, right parentheses.

Back to Figure

The window has a text label, a text box, and 2 buttons. The text label reads: Enter a value to push onto the stack. The buttons are labeled push and pop. A stack is pictorially represented in the window. The stack is represented by a stack of rectangles from top to bottom. The rectangles are numbered 9 down to 0 from top to bottom. The bottommost rectangle is marked top. Each rectangle is marked 0.

Back to Figure

The window has a text label, a text box, and 2 buttons. The text label reads: Enter a value to push onto the stack. The entry in the text box reads: 98. The buttons are labeled push and pop. A stack is pictorially represented in the window. The stack is represented by a stack of rectangles from top to bottom. The rectangles are numbered 9 down to 0 from top to bottom. The rectangles numbered from 9 down to 4 are marked 0. The markings within the rectangles numbered from 3 to 0 are as follows: 98, 12, 56, 34. Rectangle 2 is marked top. A caption to the right reads: POP was CORRECT.

Back to Figure

The table has an unmarked column followed by 2 columns labeled Index and Player object. The row entries are as follows. Row 1. Unmarked column: blank. Index: 7. Player object: blank. Row 2. Unmarked column: blank. Index: 6. Player object: blank. Row 3. Unmarked column: blank. Index: 5. Player object: blank. Row 3. Unmarked column: blank. Index: 5. Player object: blank. Row 4. Unmarked column: back. Index: 4. Player object: left parentheses, 6, comma, Steve, comma, N F L, right parentheses. Row 4 is shaded. Row 5. Unmarked column: blank. Index: 3. Player object: left parentheses, 8, comma, Gino, comma, Diablo, right parentheses. Row 6. Unmarked column: blank. Index: 2. Player object: left parentheses, 7, comma, Sarah, comma, Mario, right parentheses. Row 7. Unmarked column: blank. Index: 1. Player object: left parentheses, 2, comma, Jin, comma, Golf, right parentheses. Row 8. Unmarked column: front. Index: 0. Player object: left parentheses, 5, comma, Ajay, comma, Sonic, right parentheses. Row 8 is shaded.

Back to Figure

The table has an unmarked column followed by 2 columns labeled Index and Player object. The row entries are as follows. Row 1. Unmarked column: blank. Index: 7. Player object: blank. Row 2. Unmarked column: blank. Index: 6. Player object: blank. Row 3. Unmarked column: blank. Index: 5. Player object: blank. Row 3. Unmarked column: blank. Index: 5. Player object: blank. Row 4. Unmarked column: back. Index: 4. Player object: left parentheses, 6, comma, Steve, comma, N F L, right parentheses. Row 4 is shaded. Row 5. Unmarked column: blank. Index: 3. Player object: left parentheses, 8, comma, Gino, comma, Diablo, right parentheses. Row 6. Unmarked column: blank. Index: 2. Player object: left parentheses, 7, comma, Sarah, comma, Mario, right parentheses. Row 7. Unmarked column: front. Index: 1. Player object: left parentheses, 2, comma, Jin, comma, Golf, right parentheses. Row 7 is shaded. Row 8. Unmarked column: blank. Index: 0. Player object: left parentheses, 5, comma, Ajay, comma, Sonic, right parentheses.

Back to Figure

7 diagrams are shown. Each diagram has a circle with numbers from 0 to 7 marked around its circumference in the clockwise direction starting from the top center. In the first diagram, 0 is marked front and 7 is marked back. In the second diagram, both front and back are marked near 0. The entry corresponding to 0 is 2, Jin, Golf. In the third diagram, 0 is marked front and 1 is marked back. The entry corresponding to 1 is 5, Mike, Sonic. In the fourth diagram, 2 is marked back. The entry corresponding to 2 is 7, Sarah, Mario. In the fifth diagram, 3 is marked back. The entry corresponding to 3 is 8, John, Diablo. In the sixth diagram, 1 is marked front. In the seventh diagram, 2 is marked front.

Back to Figure

The queue is represented by a circle with numbers ranging from 0 to 7 marked around its circumference in the clockwise direction starting from the top center. In the first diagram, 0 is marked front and 7 is marked back. The description of the second diagram is as follows. The entry corresponding to 0 is 2, Jin, Golf. The entry corresponding to 1 is 5, Mike, Sonic. The entry corresponding to 2 is 7, Sarah, Mario. The entry corresponding to 3 is 8, John, Diablo. The entry corresponding to 4 is 6, Steve, N F L. The entry corresponding to 5 is 9, Joe, N B A. The entry corresponding to 6 is 12, Jim, Nim. The entry corresponding to 7 is 18, Ann, Puzzle.

Back to Figure

Each node in the list is represented by a rectangle which is partitioned into 2 parts. The list has 4 nodes. Node 1 is marked 2, Jin, Golf. Node 2 is marked 5, Ajay, Sonic. Node 3 is marked 7, Sarah, Mario. Node 4 is marked 8, Gino, Diablo. An arrow labeled head points to Node 1. The first arrow points from Node 1 to Node 2. The second arrow points from Node 2 to Node 3. The third arrow points from Node 3 to Node 4. An arrow labeled null points away from Node 4.

Back to Figure

Each node in the list is represented by a rectangle which is partitioned into 2 parts. The list has 4 nodes. Node 1 is marked 2, Jin, Golf. Node 2 is marked 5, Ajay, Sonic. Node 3 is marked 7, Sarah, Mario. Node 4 is marked 8, Gino, Diablo. An arrow labeled head points to Node 1. The first arrow points from Node 1 to Node 2. The second arrow points from Node 2 to Node 3. The third arrow points from Node 3 to Node 4. An arrow labeled null points away from Node 4. A fifth node below the 4 nodes is marked p n below. Node 5 is marked 6, Steve, N F L.

An arrow labeled null points away from Node 5.

Back to Figure

Each node in the list is represented by a rectangle which is partitioned into 2 parts. The list has 5 nodes. Node 1 is marked 2, Jin, Golf. Node 2 is marked 5, Ajay, Sonic. Node 3 is marked 7, Sarah, Mario. Node 4 is marked 8, Gino, Diablo. Node 5 is marked 6, Steve, N F L. An arrow labeled head points to Node 1. Node 2 is labeled previous. Node 3 is labeled current. The first arrow points from Node 1 to Node 2. The second arrow points from Node 2 to Node 3. The third arrow points from Node 3 to Node 4. An arrow labeled null points away from Node 4. An arrow points from Node 5 to Node 3.

Back to Figure

Each node in the list is represented by a rectangle which is partitioned into 2 parts. The list has 5 nodes. Node 1 is marked 2, Jin, Golf. Node 2 is marked 5, Ajay, Sonic. Node 3 is marked 7, Sarah, Mario. Node 4 is marked 8, Gino, Diablo. Node 5 is marked 6, Steve, N F L. An arrow labeled head points to Node 1. Node 2 is labeled previous. Node 3 is labeled current. The first arrow points from Node 1 to Node 2. The second arrow points from Node 2 to Node 5. The third arrow points from Node 5 to Node 3. The fourth arrow points from Node 3 to Node 4.

Back to Figure

Line 1: Number of items in the list, colon, 0. Line 2: blank. Line 3: inserting I d, colon, 7. Tab space. Name, colon, Sarah. Tab space. Game, colon, Mario. Line 4: Number of items in the list, colon, 1. Line 5: i d, colon, 7. Tab space. Name, colon, Sarah. Tab space. game, colon, Mario. Line 6: blank. Line 7: inserting i d, colon, 2. Tab space. name, colon, Jin. Tab space. game, colon, Golf. Line 8: Number of items in the list, colon, 2. Line 9: i d, colon, 2. Tab space. name, colon, Jin. Tab space. game, colon, Golf. Line 10: i d, colon, 7. Tab space. name, colon, Sarah. Tab space. game, colon, Mario. Line 11: blank. Line 12: inserting i d, colon, 5. Tab space. name, colon, Ajay. Tab space. game, colon, Sonic. Line 13: Number of items in the list, colon, 3. Line 14: i d, colon, 2. Tab space. name, colon, Jin. Tab space. game, colon, Golf. Line 15: i d, colon, 5. Tab space. name, colon, Ajay. Tab space. game, colon, Sonic. Line 16: i d, colon, 7. Tab space. name, colon, Sarah. Tab space. game, colon, Mario. Line 17: blank. Line 18: inserting i d, colon, 8. Tab space. name, colon, Gino. Tab space. game, colon, Diablo. Line 19: Number of items in the list, colon, 4. Line 20: i d, colon, 2. Tab space. name, colon, Jin. Tab space. game, colon, Golf. Line 21: i d, colon, 5. Tab space. name, colon, Ajay. Tab space. game, colon, Sonic. Line 22: i d, colon, 7. Tab space. name, colon, Sarah. Tab space. game, colon, Mario. Line 23: i d, colon, 8. Tab space. name, colon, Gino. Tab space. game, colon, Diablo.

Back to Figure

The window has 3 panes. Pane 1 has a text label, a text box, and 5 buttons. The text label reads: Node data. The buttons are labeled insert, delete, to String, count, and clear.

Back to Figure

The window has 3 panes. Pane 1 has a text label, a text box, and 5 buttons. The text label reads: Node data. The entry in the text box reads: 67. The buttons are labeled insert, delete, to String, count, and clear. A pictorial representation of a list with 2 nodes is shown in pane 2. Node 1 is marked 45 and Node 2 is marked 78. An arrow points from Node 1 to Node 2. The arrow pointing away from Node 2 is marked NULL. The entries in pane 3 are as follows: inserting 45, inserting 78, inserting 67, List, colon, 45, 67, 78, deleting 67.

Back to Figure

The node is represented by a rectangle which is divided into 3 parts. The center part of the rectangle is marked 5, Ajay, Sonic. 2 arrows point away from the first and third parts of the rectangle.

Back to Figure

The doubly linked list has 2 nodes. Each node is represented by a rectangle which is partitioned into 3 parts. Node 1 is marked 2, Jin, Golf. Node 2 is marked 8, Gino, Diablo. A right arrow points from Node 1 to Node 2.

A left arrow points from Node 2 to Node 1. A left arrow points away from Node 1 and a right arrow points to Node 1. A right arrow points away from Node 2. A left arrow points to Node 2.

Back to Figure

The doubly linked list has 3 nodes. Each node is represented by a rectangle which is partitioned into 3 parts. Node 1 is marked 2, Jin, Golf. Node 2 is marked 8, Gino, Diablo. A right arrow points from Node 1 to Node 2. A left arrow points from Node 2 to Node 1. A left arrow points away from Node 1 and a right arrow points to Node 1. A right arrow points away from Node 2. A left arrow points to Node 2. Node 2 is labeled current. Node 3 is labeled p n. Node 3 is marked 6, Steve, N F L. A left arrow labeled null points away from Node 3. A right arrow labeled null points away from Node 3.

Back to Figure

The doubly linked list has 3 nodes. Each node is represented by a rectangle which is partitioned into 3 parts. Node 1 is marked 2, Jin, Golf. Node 2 is marked 8, Gino, Diablo. A right arrow points from Node 1 to Node 2. A left arrow points from Node 2 to Node 1. A left arrow points away from Node 1 and a right arrow points to Node 1. A right arrow points away from Node 2. A left arrow points to Node 2. Node 2 is labeled current. Node 3 is labeled p n. Node 3 is marked 6, Steve, N F L. A left arrow labeled null points away from Node 3. A right arrow points from Node 3 to Node 2.

Back to Figure

The doubly linked list has 3 nodes. Each node is represented by a rectangle which is partitioned into 3 parts. Node 1 is marked 2, Jin, Golf. Node 2 is marked 8, Gino, Diablo. A right arrow points from Node 1 to Node 2. A left arrow points from Node 2 to Node 1. A left arrow points away from Node 1 and a right arrow points to Node 1. A right arrow points away from Node 2. A left arrow points to Node 2. Node 2 is labeled current. Node 3 is labeled p n. Node 3 is marked 6, Steve, N F L. A left arrow labeled null points away from Node 3. A right arrow points from Node 3 to Node 2. A right arrow points from Node 1 to Node 3.

Back to Figure

The doubly linked list has 3 nodes. Each node is represented by a rectangle which is partitioned into 3 parts. Node 1 is marked 2, Jin, Golf. Node 2 is marked 8, Gino, Diablo. A right arrow points from Node 1 to Node 2. A left arrow points from Node 2 to Node 1. A left arrow points away from Node 1 and a right arrow points to Node 1. A right arrow points away from Node 2. A left arrow points to Node 2. Node 2 is labeled current. Node 3 is labeled p n. Node 3 is marked 6, Steve, N F L. A left arrow points from Node 3 to Node 1. A right arrow points from Node 3 to Node 2. A right arrow points from Node 1 to Node 3.

Back to Figure

The doubly linked list has 3 nodes. Each node is represented by a rectangle which is partitioned into 3 parts. Node 1 is marked 2, Jin, Golf. Node 2 is marked 8, Gino, Diablo. A left arrow points away from Node 1 and a right arrow points to Node 1. A right arrow points away from Node 2. A left arrow points to Node 2. Node 2 is labeled current. Node 3 is labeled p n. Node 3 is marked 6, Steve, N F L. A left arrow points from Node 3 to Node 1. A right arrow points from Node 3 to Node 2. A right arrow points from Node 1 to Node 3. A left arrow points from Node 2 to Node 3.

Back to Figure

The doubly linked list has 3 nodes. Each node is represented by a rectangle which is partitioned into 3 parts. Node 1 is marked 2, Jin, Golf. Node 2 is marked 7, Sarah, Mario. Node 3 is marked 8, Gino, Diablo. A left arrow points away from Node 1. A right arrow points to Node 1. A right arrow points from Node 1 to Node 2. A left arrow points from Node 2 to Node 1. A right arrow points from Node 2 to Node 3. A left arrow points from Node 3 to Node 2. A right arrow points away from Node 3. A left arrow points to Node 2. Node 2 is labeled current.

Back to Figure

The doubly linked list has 3 nodes. Each node is represented by a rectangle which is partitioned into 3 parts. Node 1 is marked 2, Jin, Golf. Node 2 is marked 7, Sarah, Mario. Node 3 is marked 8, Gino, Diablo. A left arrow points away from Node 1. A right arrow points to Node 1. A left arrow points from Node 2 to Node 1. A right arrow points from Node 2 to Node 3. A left arrow points from Node 3 to Node 2. A right arrow points away from Node 3. A left arrow points to Node 2. A right arrow points from Node 1 to Node 3. Node 2 is labeled current.

Back to Figure

The doubly linked list has 3 nodes. Each node is represented by a rectangle which is partitioned into 3 parts. Node 1 is marked 2, Jin, Golf. Node 2 is marked 7, Sarah, Mario. Node 3 is marked 8, Gino, Diablo. A left arrow points away from Node 1. A right arrow points to Node 1. A left arrow points from Node 2 to Node 1. A right arrow points from Node 2 to Node 3. A left arrow points from Node 3 to Node 2. A right arrow points away from Node 3. A left arrow points to Node 2. A right arrow points from Node 1 to Node 3. A left arrow points from Node 3 to Node 1. Node 2 is labeled current and it is also highlighted.

Back to Figure

Line 1: Number of items in the list, colon, 0. Line 2: blank. Line 3: Number of items in the list, colon, 1. Line 4: i d, colon, 7. Tab space. name, colon, Sarah. Tab space. game, colon, Mario. Line 5: blank. Line 6: Number of items in the list, colon, 2. Line 7: i d, colon, 2. Tab space. name, colon, Jin. Tab space. game, colon, Golf. Line 8: i d, colon, 7. Tab space. name, colon, Sarah. Tab space. game, colon, Mario. Line 9: blank. Line 10: Number of items in the list, colon, 3. Line 11: i d, colon, 5. Tab space. name, colon, Ajay. Tab space. game, colon, Sonic. Line 12: i d, colon, 2. Tab space. name, colon, Jin. Tab space. game, colon, Golf. Line 13: i d, colon, 7. Tab space. name, colon, Sarah. Tab space. game, colon, Mario. Line 14: blank. Line 15: Player successfully deleted, colon. Line 16: Number of items in the list, colon, 2. Line 17: i d, colon, 5. Tab space. name, colon, Ajay. Tab space. game, colon, Sonic. Line 18: i d, colon, 7. Tab space. name, colon, Sarah. Tab space. game, colon, Mario. Line 19: blank. Line 20: Player successfully deleted, colon. Line 21: Number of items in the list, colon, 1. Line 22: i d, colon, 7. Tab space. name, colon, Sarah. Tab space. game, colon, Mario.

Back to Figure

A large rectangle is divided into 2 smaller rectangles. The first rectangle is labeled, first, an item. The second rectangle is labeled, rest, a linked list.

Back to Figure

A large rectangle is divided into 2 small rectangles labeled first and rest. The rectangle labeled first is marked p 1. The rectangle labeled rest is marked r 1.

Back to Figure

A large rectangle is divided into 2 small rectangles labeled first and rest. The rectangle labeled first is marked p. The rectangle labeled rest has 2 rectangles labeled p 1 and r 1.

Back to Figure

A rectangle is divided into a small rectangle labeled first and a large rectangle labeled rest. The small rectangle is marked p. The rectangle labeled rest has a rectangle which is divided into a small rectangle marked p 1 and a large rectangle marked r 1.

Back to Figure

A rectangle is divided into a small rectangle labeled first and a large rectangle labeled rest. The small rectangle is marked p 1. The large rectangle has 2 small rectangles which are marked p 1 and r 1.

Back to Figure

A rectangle is divided into a small rectangle labeled first and a large rectangle labeled rest. Rectangle first is marked p 1 and rectangle rest is marked r 1.

Back to Figure

Line 1: The list is. Line 2: empty. Line 3: blank. Line 4: Inserting i d, colon, 7. Tab space. Name, colon, Sarah. Game, colon, Mario. Line 5: The list is. Line 6: i d, colon, 7. Tab space. Name, colon, Sarah. Tab space. Game, colon, Mario. Line 7: blank. Line 8: Inserting i d, colon, 2. Tab space. name, colon, Jin. game, colon, Golf. Line 9: The list is. Line 10: i d, colon, 2. name, colon, Jin. game, colon, Golf. Line 11: i d, colon, 7. name, colon, Sarah. game, colon, Mario. Line 12: blank. Line 13: Inserting i d, colon, 5. Tab space. name, colon, Ajay. game, colon, Sonic. Line 14: The list is. Line 15: i d, colon, 5. name, colon, Ajay. game, colon, Sonic. Line 16: i d, colon, 2. name, colon, Jin. game, colon, Golf. Line 17: i d, colon, 7. name, colon, Sarah. game, colon, Mario. Line 18: blank. Line 19: Player deleted, colon, i d, colon, 2. name, colon, Jin. game, colon, Golf. Line 20: The list is. Line 21: i d, colon, 5. name, colon, Ajay. game, colon, Sonic. Line 22: i d, colon, 7. name, colon, Sarah. game, colon, Mario. Line 23: blank. Line 24: Player deleted i d, colon, 7. name, colon, Sarah. game, colon, Mario. Line 25: The list is. Line 26: i d, colon, 5. name, colon, Ajay. game, colon, Sonic. Line 27: blank. Line 28: The list is. Line 29: i d, colon, 5. name, colon, Ajay. game, colon, Sonic. Line 30: blank. Line 31: Player deleted, colon, i d, colon, 5. name, colon, Ajay. game, colon, Sonic. Line 32: The list is. Line 33: empty. Line 34: blank. Line 35: The list is. Line 36: empty.

Back to Figure

f, left parentheses, n, right parentheses, equals, 3 asterisk n squared plus 6 asterisk n plus 12.

Back to Equation

f, left parentheses, n, right parentheses, equals, 3 asterisk n squared plus 6 asterisk n plus 12, greater than or equal to, 3 asterisk n squared.

Back to Equation

f, left parentheses, n, right parentheses, equals, n squared, asterisk, left parentheses, 3 plus 6 forward slash n plus 12 forward slash n squared, right parentheses.

Back to Equation

6 forward slash n, less than or equal to, 1. 12, forward slash, n squared, less than 1.

Back to Equation

Line 1: f, left parentheses, n, right parentheses, equals, a subscript p, n superscript P, plus, a subscript p minus 1, n superscript, p minus 1, plus, dot, dot, dot, plus, a subscript 2, n superscript 2, plus, a subscript 1 n, plus, a subscript 0, where a subscript p is strictly positive. Line 2: f, left parentheses, n, right parentheses, equals, a subscript P, n superscript P, left parentheses, 1 plus, left parentheses, a subscript p minus 1, forward slash, a subscript p, right parentheses, 1, forward slash, n, plus, dot, dot, dot, plus, left parentheses, a subscript 2, forward slash, a subscript p, right parentheses, 1, forward slash, n superscript p minus 2, plus, left parentheses, a subscript 1, forward slash, a subscript p, right parentheses, 1, forward slash, n superscript p minus 1, plus, left parentheses, a subscript p, forward slash, a subscript 0, right parentheses, 1, forward slash, n superscript P, right parentheses. Line 3: f, left parentheses, n, right parentheses, less than or equal to, a subscript p, n superscript p, left parentheses, 1, plus, vertical bar, left parentheses, a subscript p minus 1, forward slash, a subscript p, right parentheses, vertical bar, 1, forward slash, n, plus, dot, dot, dot, plus, vertical bar, left parentheses, a subscript 2, forward slash, a subscript p, right parentheses, vertical bar, 1, forward slash, n superscript p minus 2, plus, vertical bar, left parentheses, a subscript 1, forward slash, a subscript p, right parentheses, vertical bar, 1, forward slash, n superscript p minus 1, plus, vertical bar, left parentheses, a subscript p, forward slash, a subscript 0, right parentheses, vertical bar, 1, forward slash, n superscript p, right parentheses.

Back to Equation

Line 1: f, left parentheses, n, right parentheses, less than or equal to, a subscript p, n superscript p, left parentheses, 1 plus M 1 forward slash n, plus, dot, dot, dot, plus, M 1, forward slash, n superscript p minus 2, plus, M 1 forward slash n superscript p minus 1 plus M 1 forward slash n superscript p, right parentheses. Line 2: f, left parentheses, n, right parentheses, less than or equal to, a subscript p, n superscript p, left parentheses, 1 plus

M, left parentheses, 1 forward slash n, plus, dot, dot, dot, plus, 1 forward slash n superscript p minus 2, plus, 1 forward slash n superscript p minus 1 plus 1 forward slash n superscript p, right parentheses, right parentheses. Line 3: f, left parentheses, n, right parentheses, less than or equal to, a subscript p, n superscript p, left parentheses, 1 plus M, left parentheses, negative 1 plus 1 plus 1 forward slash n, plus, dot, dot, dot, plus, 1 forward slash n superscript p minus 2, plus, 1 forward slash, n superscript p minus 1, plus, 1 forward slash, n superscript p, right parentheses, right parentheses.

Back to Equation

Line 1: f, left parentheses, n, right parentheses, less than or equal to, a subscript p, n superscript p, left parentheses, 1 plus M, left parentheses, negative 1, plus, left parentheses, 1 minus 1 forward slash n superscript p plus 1, right parentheses, forward slash, left parentheses, 1 minus 1 forward slash n, right parentheses, right parentheses, right parentheses. Line 2: f, left parentheses, n, right parentheses, less than or equal to, s subscript p, n superscript p, left parentheses, 1 plus M, left parentheses, negative 1, plus, left parentheses, 1 minus 1 forward slash n superscript p plus 1, right parentheses, asterisk, left parentheses, n forward slash, left parentheses, n minus 1, right parentheses, right parentheses, right parentheses, right parentheses.

Back to Equation

Line 1: f, left parentheses, n, right parentheses, less than or equal to, a subscript p, n superscript p, left parentheses, 1 plus M, left parentheses, negative 1, plus, left parentheses, n forward slash, left parentheses, n minus 1, right parentheses, right parentheses, right parentheses, right parentheses. Line 2: f, left parentheses, n, right parentheses, less than or equal to, a subscript p, n superscript p, left parentheses, 1 plus M, left parentheses, left parentheses, negative n plus 1 plus n, right parentheses, forward slash, left parentheses, n minus 1, right parentheses, right parentheses, right parentheses. Line 3: f, left parentheses, n, right parentheses, less than or equal to, a subscript p, n superscript p, left parentheses, 1 plus M, left parentheses, 1 forward slash, left parentheses, n minus 1, right parentheses, right parentheses, right parentheses.

Back to Equation

f, left parentheses, n, right parentheses, less than or equal to, a subscript p, n superscript p, left parentheses, 1 plus M, right parentheses, for n greater than or equal to 2.

Back to Equation

f, left parentheses, n, right parentheses, less than or equal to, c subscript 0, n superscript p.

Back to Equation

The table has 2 columns labeled Statement and hash

Times Executed. The row entries are as follows. Row 1. Statement: left parentheses, 1, right parentheses. Hash Times Executed: 1. Row 2. Statement: left parentheses, 2, right parentheses. Hash Times Executed: 1. Row 3. Statement: left parentheses, 3, right parentheses. Hash Times Executed: n plus 1. Row 4. Statement: left parentheses, 4, right parentheses. Hash Times Executed: n. Row 5. Statement: left parentheses, 5, right parentheses. Hash Times Executed: n. Row 6. Statement: left parentheses, 6, right parentheses. Hash Times Executed: 1.

Back to Table

Line 1: T, left parentheses, n, right parentheses, equals, 1 plus 1 plus, left parentheses, n plus 1, right parentheses, plus, n plus n plus 1. Line 2: Equals sign, 3 n plus 4. Line 3: Equals sign, O, left parentheses, n, right parentheses.

Back to Equation

The table has 2 columns labeled Statement and number of times executed. The row entries are as follows. Row 1. Statement: left parentheses, 1, right parentheses. Number of Times Executed: 1. Row 2. Statement: left parentheses, 2, right parentheses. Number of Times Executed: 1 plus, left parentheses, n plus 1, right parentheses, plus, n, equals, 2 asterisk n, plus, 2. Row 3. Statement: left parentheses, 3, right parentheses. Number of Times Executed: n asterisk, left parentheses, 1 plus, left parentheses, n plus 1, right parentheses, plus, n, right parentheses, equals, 2 asterisk n squared plus 2 asterisk n. Row 4. Statement: left parentheses, 4, right parentheses. Number of Times Executed: n asterisk n equals n squared. Row 5. Statement: left parentheses, 5, right parentheses. Number of Times Executed: between 0 and n asterisk n. Row 6. Statement: left parentheses, 6, right parentheses. Number of Times Executed: 1.

Back to Table

The equation is as follows. T, left parentheses, n, right parentheses, equals, 1 plus, left parentheses, 2 asterisk n plus 2, right parentheses, plus, left parentheses, 2 asterisk n squared plus 2 asterisk n, right parentheses, plus, left parentheses, n squared, right parentheses, plus, x plus, 1. This equals 3 asterisk n squared plus 4 asterisk n plus 4 plus x.

Back to Equation

The equations are as follows. Line 1: 3 asterisk n squared, plus, 4 asterisk n, 4, left angled bracket, equals sign, T, left parentheses, n, right parentheses, left angled bracket, equals sign, 3 asterisk n squared, plus, 4 asterisk n, plus, 4, plus, n squared. Line 2: 3 asterisk n squared, plus, 4 asterisk n, plus, 4, left angled bracket, equals sign, T, left parentheses, n, right parentheses, left angled bracket, equals sign, 4 asterisk n squared, plus, 4 asterisk n, plus 4.

Back to Equation

The table has 2 columns labeled Statement and number of times executed. The row entries are as follows. Row 1. Statement: left parentheses, 1, right parentheses. Number of times executed: 1 plus, left parentheses, between 1 and left parentheses, n plus 1, right parentheses, right parentheses plus left parentheses, between 0 and n, right parentheses. Row 2. Statement: left parentheses, 2, right parentheses. Number of times executed: between 1 and n. Row 3. Statement: left parentheses, 3, right parentheses. Number of times executed: 0 or 1. Row 4. Statement: left parentheses, 4, right parentheses. Number of times executed: 1 or 0.

Back to Table

The equations are as follows. Line 1: 1 plus, left parentheses, 1, right parentheses, plus, left parentheses, 0, right parentheses, plus, 1, plus, 1, left angled bracket, equals, T, left parentheses, n, right parentheses, left angled bracket, equals sign, 1 plus, left parentheses, n plus 1, right parentheses, plus, n plus n plus 1. Line 2: 4, left angled bracket, equals sign, T, left parentheses, n, right parentheses, left angled bracket, equals sign, 3 n plus 3.

Back to Equation

Line 1: T, left parentheses, n, right parentheses, equals, 1, plus, left parentheses, n plus 1, right parentheses, forward slash, 2, plus, n forward slash 2, plus, n forward slash 2, plus, 1. Line 2: Equals sign, 3 n, forward slash, 2, plus, 2 and a half. Line 3: equals sign, O, left parentheses, n, right parentheses.

Back to Equation

Line 1: T 1, left parentheses, n, right parentheses, equals, 1 plus T 1, left parentheses, n minus 1, right parentheses, plus 1. Line 2: equals sign, T 1, left parentheses, n minus 1, right parentheses, plus, 2. 2 forward slashes. Equation 14.1.

Back to Equation

T 1, left parentheses, x, right parentheses, equals, T 1, left parentheses, x minus 1, right parentheses, plus, 2, where x is some integer. 2 forward slashes. Equation 14.2.

Back to Equation

T 1, left parentheses, n minus 1, right parentheses, equals, T 1, left parentheses, n minus 2, right parentheses, plus, 2.

Back to Equation

T 1, left parentheses, n, right parentheses, equals, T 1, left parentheses, n minus 2, right parentheses, plus, 2, plus, 2. Line 2: Equals sign, T 1, left parentheses, n minus 2, right parentheses, plus, 2 asterisk 2. 2 forward slashes.

Equation 14.3.

Back to Equation

The equation reads: T 1, left parentheses, n minus 2, right parentheses, equals, T 1, left parentheses, n minus 3, right parentheses, plus, 2.

Back to Equation

The equation is as follows. Line 1: T 1, left parentheses, n, right parentheses, equals, T 1, left parentheses, n minus 3, right parentheses, 2, plus, 2 asterisk 2. Line 2: equals sign, T 1, left parentheses, n minus 3, right parentheses, plus, 2 asterisk 3. 2 forward slashes, Equaiton 14.4.

Back to Equation

T 1, left parentheses, n minus 3, right parentheses, equals, T 1, left parentheses, n minus 4, right parentheses, plus, 2.

Back to Equation

Line 1: T 1, left parentheses, n, right parentheses, equals, T 1, left parentheses, n minus 4, right parentheses, plus, 2, plus, 2 asterisk 3. Line 2: Equals sign, T 1, left parentheses, n minus 4, right parentheses, plus, 2 asterisk 4.

Back to Equation

T 1, left parentheses, n, right parentheses, equals, T 1, left parentheses, n minus k, right parentheses, plus, 2 asterisk k, where k is an integer between 1 and n. 2 forward slashes. Equation 14.5.

Back to Equation

T 1, left parentheses, n, right parentheses, equals, T 1, left parentheses, 0, right parentheses, plus, 2 asterisk n, equals, 1 plus 2 asterisk n, equals, 2 asterisk n plus 1.

Back to Equation

Line 1: T 1, left parentheses, 0, right parentheses, equals, 2 asterisk 0, plus, 1. Line 2: Equals sign, 1.

Back to Equation

T 1, left parentheses, n plus 1, right parentheses, equals, T 1, left parentheses, n, right parentheses, plus, 2.

Back to Equation

Line 1: T 1, left parentheses, n plus 1, right parentheses, equals, 2 asterisk n, plus, 1, plus, 2. Line 2: Equals sign, 2 asterisk n, plus, 2, plus, 1. Line 3: Equals sign, 2 asterisk, left parentheses, n plus 1, right parentheses, plus 1.

Back to Equation

Line 1: T 2, left parentheses, n, right parentheses, equals, 1, plus, T 2, left parentheses, n minus 1, right parentheses, plus, T 2, left parentheses, n minus 1, right parentheses, plus, 1. Line 2: Equals sign, 2 asterisk, T 2, left parentheses, n minus 1, right parentheses, plus, 2. 2 forward slashes. Equation 14.6.

Back to Equation

The equation is as follows. T 2, left parentheses, x, right parentheses, equals, 2 asterisk T 2, left parentheses, x minus 1, right parentheses, plus, 2. 2 forward slashes. Equation 14.7.

Back to Equation

The equation is as follows. T 2, left parentheses, n minus 1, right parentheses, 2 asterisk T 2, left parentheses, n minus 2, right parentheses, plus, 2.

Back to Equation

Line 1: T 2, left parentheses, n, right parentheses, equals, 2 asterisk, left parentheses, 2 asterisk T 2, left parentheses, n minus 2, right parentheses, plus, 2, right parentheses, plus, 2. Line 2: Equals sign. 2 to the power of 2 asterisk T 2, left parentheses, n minus 2, right parentheses, plus, 2 squared, plus 2. 2 forward slashes. Equation 14.8.

Back to Equation

The equation is as follows. T 2, left parentheses, n minus 2, right parentheses, equals, 2 asterisk T 2, left parentheses, n minus 3, right parentheses, plus, 2.

Back to Equation

Line 1: T 2, left parentheses, n, right parentheses, equals, 2 squared asterisk, left parentheses, 2 asterisk T 2, left parentheses, n minus 3, right parentheses, plus, 2, right parentheses, plus, 2 squared, plus, 2. Line 2: 2 cubed asterisk T 2, left parentheses, n minus 3, right parentheses, plus, 2 cubed, plus, 2 squared, plus, 2. 2 forward slashes. Equation 14.9.

Back to Equation

The equation is as follows. T 2, left parentheses, n minus 3, right parentheses, equals, 2 asterisk, T 2, left parentheses, n minus 4, right parentheses, plus, 2.

Back to Equation

Line 1: T 2, left parentheses, n, right parentheses, equals, 2 cubed, asterisk, left parentheses, 2 asterisk T 2, left parentheses, n minus 4, right parentheses, plus, 2, right parentheses, plus, 2 cubed, plus, 2 squared, plus, 2. Line 2: Equals sign. 2 to the power of 4, asterisk, T 2, left parentheses, n minus 4, right parentheses, plus, 2 to the power of 4, plus, 2 cubed, plus, 2 squared, plus, 2. 2 forward slashes. Equation 14.10.

Back to Equation

The equation is as follows. T 2, left parentheses, n, right parentheses, equals, 2 to the power of k, asterisk, T 2, left parentheses, n minus k, right parentheses, plus, 2 to the power of k, plus, 2 to the power of k minus 1, plus, dot, dot, dot, plus, 2 squared, plus, 2, where k is an integer between 1 and n. 2 forward slashes. Equation 14.11.

Back to Equation

Line 1: 2 to the power of k, plus, 2 to the power of k minus 1, plus, dot, dot, dot, plus, 2 to the power of 2, plus, 2, equals, negative 1, plus, 2 to the power of k, plus, 2 to the power of k minus 1, plus, dot, dot, dot, plus, 2 to the power of 2, plus, 2, plus, 1. Line 2: Equals sign. negative 1, plus, left parentheses, 2 to the power of k plus 1, minus, 1, right parentheses, forward slash, left parentheses, 2 minus 1, right parentheses. Line 3: Equals sign. 2 to the power of k plus 1, minus, 2.

Back to Equation

The equation is as follows. T 2, left parentheses, n, right parentheses, equals, 2 to the power of k, asterisk, T 2, left parentheses, n minus k, right parentheses, plus, 2 to the power of k plus 1, minus, 2, where k is an integer between 1 and n. 2 forward slashes. Equation 14.12.

Back to Equation

Line 1: T 2, left parentheses, n, right parentheses, equals, 2 to the power of n, asterisk, T 2, left parentheses, 0, right parentheses, plus, 2 to the power of n plus 1, minus, 2.

Line 2: Equals sign. 2 to the power of n, asterisk, 1, plus, 2 to the power of n plus 1, minus, 2. Line 3: Equals sign. 2 to the power of n, plus, 2 to the power of n plus 1, minus, 2. Line 4: Equals sign. 2 to the power of n, left parentheses, 1 plus 2, right parentheses, minus, 2. Line 5: Equals sign. 3 asterisk 2 to the power of n, minus, 2. Line 6: Equals sign. O, left parentheses, 2 to the power of n, right parentheses.

Back to Equation

The window has a text label, a text box, 2 buttons, 2 captions and a bar graph. The text label reads: Enter an integer between 5 and 25 for the number of array elements. The entry within the text box reads: 15. The 2 buttons are inactive. The button labels are as follows: Show Array and Selection Sort. The 2 captions below the buttons are as follows: Sorting using Selection Sort. Your number of steps, equals, 47. The heights of the bars in the bar graph are as follows: 72, 59, 69, 64, 66, 67, 74, 65 71, 71, 60, 56, 74, 74, and 76. The seventh bar of height 74 is marked red. The ninth bar of height 71 is white with a red outline.

Back to Figure

Line 1: T, left parentheses, n, right parentheses, equals, 1 plus 2 plus 1 plus 1 plus T, left parentheses, n forward slash 2, right parentheses. Line 2: Equals sign. T, left parentheses, n forward slash 2, right parentheses, plus, 5.

2 forward slashes. Equation 14.13.

Back to Equation

Line 1: T, left parentheses, n forward slash 2, right parentheses, equals, T, left parentheses, left parentheses, n forward slash 2, right parentheses, forward slash, 2, right parentheses, plus, 5. Line 2: Equals sign. T, left parentheses, n forward slash 2 squared, right parentheses, plus, 5.

Back to Equation

Line 1: T, left parentheses, n, right parentheses, equals, left parentheses, T, left parentheses, n forward slash 2 squared, right parentheses, plus, 5, right parentheses, plus, 5. Line 2: Equals sign. T, left parentheses, n forward slash 2 squared, right parentheses, plus, 5 asterisk 2. 2 forward slashes. Equation 14.15.

Back to Equation

Line 1: T, left parentheses, n forward slash 2 squared, right parentheses, equals, T, left parentheses, left parentheses, n forward slash 2 squared, right parentheses, forward slash, 2, right parentheses, plus, 5. Line 2: Equals sign. T, left parentheses, n forward slash 2 cubed, right parentheses, plus, 5.

Back to Equation

Line 1: T, left parentheses, n, right parentheses, equals, left parentheses, T, left parentheses, n forward slash, 2 cubed, right parentheses, plus, 5, right parentheses, plus, 5 asterisk 2. Line 2: equals sign. T, left parentheses, n forward slash 2 cubed, right parentheses, plus, 5 asterisk 3. 2 forward slashes. Equation 14.16.

Back to Equation

Line 1: T, left parentheses, n forward slash 2 cubed, right parentheses, equals, T, left parentheses, left parentheses, n forward slash 2 cubed, right parentheses, forward slash, 2, right parentheses, plus, 5. Line 2: Equals sign. T, left parentheses, n forward slash 2 to the power of 4, right parentheses, plus, 5.

Back to Equation

Line 1: T, left parentheses, n, right parentheses, equals, left parentheses, T, left parentheses, n forward slash 2 to the power of 4, right parentheses, plus, 5, right parentheses, plus, 5 asterisk 3. Line 2: Equals sign. T, left parentheses, n forward slash 2 to the power of 4, right parentheses, plus, 5 asterisk 4. 2 forward slashes. Equation 14.17.

Back to Equation

T, left parentheses, n, right parentheses, equals, T, left parentheses, n forward slash 2 to the power of k, right parentheses, plus, 5 asterisk k.

Back to Equation

Line 1: log n, equals, log 2 to the power of k. Line 2: Equals sign, k log 2. Line 3: Equals sign, k asterisk 1. Line 4: Equals sign, k.

Back to Equation

Line 1: T, left parentheses, n, right parentheses, equals, T, left parentheses, 1, right parentheses, plus, 5 asterisk log n. Line 2: Equals sign, 2 plus 5 asterisk log n. Line 3: Equals sign, O, left parentheses, log n, right parentheses.

Back to Equation

Line 1: T, left parentheses, n, right parentheses, equals, T, left parentheses, n forward slash 2, right parentheses, plus, T, left parentheses, n forward slash 2, right parentheses, plus, n. Line 2: Equals sign, 2 T, left parentheses, n forward slash 2, right parentheses, plus, n.

Back to Equation

Line 1: T, left parentheses, n, right parentheses, equals, 2 T, left parentheses, n forward slash 2, right parentheses, plus, n. Line 2: Equals sign, 2, left parentheses, 2 T, left parentheses, n forward slash 2 squared, right parentheses, plus, n forward slash 2, right parentheses, plus n. Line 3: Equals sign. 2 squared, T, left parentheses, n forward slash 2 squared, right parentheses, plus, 2 n.

Back to Equation

The first equation is as follows. Line 1: T, left parentheses, n, right parentheses, equals, 2 squared T, left parentheses, n forward slash 2 squared, right parentheses, plus, 2 n. Line 2: Equals sign. 2 squared, left parentheses, 2 T, left parentheses, n forward slash 2 cubed, right parentheses, plus, n forward slash 2 squared, right parentheses, plus, 2 n. Line 3: Equals sign. 2 cubed T, left parentheses, n forward slash 2 cubed, right parentheses, plus, 3 n. The second equation is as follows. Line 1: T, left parentheses, n, right parentheses, equals, 2 cubed T, left parentheses, n forward slash 2 cubed, right parentheses, plus, 3 n. Line 2: Equals sign. 2 cubed, left parentheses, 2 T, left parentheses, n forward slash 2 cubed, right parentheses, plus, n forward slash 2 cubed, right parentheses, plus, 3 n. Line 3: Equals sign, 2 to the power of 4 T, left parentheses, n forward slash 2 to the power of 4, right parentheses, plus 4 n.

Back to Equation

T, left parentheses, n, right parentheses, equals, 2 to the power of k, T, left parentheses, n, forward slash, 2 to the power of k, right parentheses, plus, k n.

Back to Equation

Line 1: T, left parentheses, n, right parentheses, equals, n T, left parentheses, 1, right parentheses, plus, n log n. Line 2: Equals sign. O, left parentheses, n log n, right parentheses.

Back to Equation

Line 1: 11, 379, 481, 452, 531, 710, 1118, 1169, 1177, dot, dot, dot. Line 2: Unsorted array, colon, Insertion sort count, equals, 259873. Line 3: 11, 379, 418, 452, 531, 710, 1118, 1169, 1177, dot, dot, dot. Line 4: Sorted array, colon, Insertion sort count, equals, 1023. Line 5: 43, 471, 589, 643, 653, 845, 1050, 1065, 1089, dot, dot, dot. Line 6: Unsorted array, colon, merge sort count, equals, 8917. Line 7: 43, 471, 589, 643, 653, 845, 1050, 1065, 1089, dot, dot, dot. Line 8: Sorted array, colon, Merge sort count, equals, 5120. Line 9: 31, 38, 77, 192, 222, 252, 547, 560, 671, 673, dot, dot, dot. Line 10: Unsorted array, colon, Quick sort count, equals, 10332. Line 11: 31, 38, 77, 192, 222, 252, 547, 560, 671, 673, dot, dot, dot. Line 12: Sorted array, colon, Quick sort count, equals, 523776.

Back to Figure

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: public static int foo 1, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, n greater than 1, right parentheses. Line 4, indented twice: return, left parentheses, 2 asterisk foo 1, left parentheses, n forward slash 4, right parentheses, right parentheses, semicolon. Line 5, indented once: else. Line 6, indented twice: return 1, semicolon. Line 7: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, void, int, for. Line 1: public static void foo 2, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int i, equals, n, semicolon, i greater than 0, semicolon, i minus, minus, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: for, left parentheses, int j equals 0, semicolon, j less than n, semicolon, j plus, plus, right parentheses. Line 6, indented twice: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 7, indented once: right brace. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, void, in, for. Line 1: public static void foo 3, left parentheses, int n, right parentheses. Line 2: Left brace. Line 3, indented once: for, left parentheses, int i equals 0, semicolon, i less than n, semicolon, i plus, plus, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: for, left parentheses, int j equals 0, semicolon, j less than i, j plus, plus, right parentheses. Line 6, indented thrice: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 7, indented once: right brace. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, void, int, for. Line 1: public static void foo 4, left parentheses, int, left square bracket, right square bracket, left square bracket, right square bracket, a r r, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int i equals 0, semicolon, i less than a r r dot length, semicolon, i plus, plus, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: for, left parentheses, int j equals a r r, left square bracket, i, right square bracket, dot, length minus 1, semicolon, j greater than or equal to 0, semicolon, j minus, minus, right parentheses. Line 6, indented thrice: System dot out dot print l n, left parentheses, double quotes, Hello world, double quotes, right parentheses, semicolon. Line 7, indented once: right brace. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, void, int, for. Line 1: public static void foo 5, left parentheses, int, left square bracket, right square bracket, left square bracket, right square bracket, left square bracket, right square bracket, a r r, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int i equals 0, semicolon, i less than a r r dot length, semicolon, i plus, plus, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: for, left parentheses, int j equals 0, semicolon, j less than a r r, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 6, indented twice: left brace. Line 7, indented thrice: for, left parentheses, int k equals 0, semicolon, k less than a r r, left square bracket, i, right square bracket, left square bracket, j, right square bracket, dot length, semicolon, k plus, plus, right parentheses. Line 8, indented thrice: System dot out dot print l n, left parentheses, double quotes, Hello World, double quotes, right parentheses, semicolon. Line 9, indented twice: right brace. Line 10, indented once: right brace. Line 11: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, void, int, if, else. Line 1: public static void foo 6, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, n less than or equal to 0, right parentheses. Line 4, indented twice: System dot out dot print l n, left parentheses, double quotes, Hello world, double quotes, right parentheses, semicolon. Line 5, indented once: else. Line 6, indented twice: foo 6, left parentheses, n minus 1, right parentheses, semicolon. Line 7: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: public static int foo 7, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, n is guaranteed to be greater than or equal to 0. Line 4, indented once: if, left parentheses, n, equals, equals, 0, right parentheses. Line 5, indented twice: return 0, semicolon. Line 6, indented once: else. Line 7, indented twice: return, left parentheses, n plus foo 7, left parentheses, n minus 1, right parentheses, right parentheses, semicolon. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: public static int foo 8, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, n is guaranteed to be greater than or equal to 1. Line 4, indented once: if, left parentheses, n, equals, equals, 1, 2 pipes, n, equals, equals, 2, right parentheses. Line 5, indented twice: return 1, semicolon. Line 6, indented once: else. Line 7, indented twice: return, left parentheses, foo 8, left parentheses, n minus 1, right parentheses, plus, foo 8, left parentheses, n minus 2, right parentheses, right parentheses, semicolon. Line 9: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, void, int, if, else. Line 1: public static void foo 9, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, n is guaranteed to be greater than or equal to 0. Line 4, indented once: if, left parentheses, n, equals, equals, 0, right parentheses. Line 5, indented twice: System dot out dot print l n, left parentheses, double quotes, done, double quotes, right parentheses, semicolon. Line 6, indented once: else. Line 7, indented twice: foo 9, left parentheses, n forward slash 2, right parentheses, semicolon. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, void, int, if, else. Line 1: public static void foo 10, left parentheses, int n, comma, int p, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, n and p are guaranteed to be greater than or equal to 1. Line 4, indented once: if, left parentheses, p greater than or equal to n, right parentheses. Line 5, indented twice: System dot out dot print l n, left parentheses, double quotes, done, double quotes, right parentheses, semicolon. Line 6, indented once: else. Line 7, indented twice: foo 10, left parentheses, n, comma, 2 asterisk p, right parentheses, semicolon. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords:

public, static, void, int, if, else, return. Line 1: public static void foo 11, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, n is guaranteed to be greater than or equal to 0. Line 4, indented once: if, left parentheses, n, equals, equals, 0, right parentheses. Line 5, indented twice: return 0, semicolon. Line 6, indented once: else. Line 7, indented twice: return, left parentheses, 5 plus 2 asterisk foo 11, left parentheses, n minus 1, right parentheses, right parentheses, semicolon. Line 8: right brace.

Back to Code

Fib, left parentheses, 1, right parentheses, equals, Fib, left parentheses, 2, right parentheses, equals, 1. Fib, left parentheses, n, right parentheses, equals, Fib, left parentheses, n minus 1, right parentheses, plus, Fib, left parentheses, n minus 2, right parentheses.

Back to Equation

H T M L code converted to a tree by a web browser. The root node is h t m l. This node has the left child, head and the right child, body. The head nod has the left child, title. The body node has the left child, p and the right child, o l. The o l node has the three l i child nodes.

Back to Figure

A tree that stores integers. The root node is 33. This node has the three child nodes, 20, 54, and 21. The node 20 has the child nodes 34 and 87. The node 34 has the child node, 17. The node 87 has the child nodes, 25, 44, 22, and 78. The node 22 has the child node, 99. The node 21 has the child node, 84.

Back to Figure

A binary tree storing integers. The root node is 33. The root node has the child nodes, 20 and 21. The node 20 has the child nodes, 34 and 87. The node 34 has the child node 17. The node 87 has the child nodes, 25 and 44. The node 21 has the child node, 84.

Back to Figure

A binary tree storing integers. Root node is 20. It has the next sibling, 54 and left child, 34.

Back to Figure

A binary tree storing integers. The root node is 33. It has a left child, 20. The node 20 has a left child, 34 and a next sibling, 54. The node 54 has a next sibling, 21. The node 34 has the left child, 17 and the next sibling 87. The node 87 hast he left child 25. The node 25 has the next sibling 44. The node 44 has the next sibling 22. The node 22 has the left child, 99 and the next sibling 78. The node 21 has the left child, 84.

Back to Figure

A complete binary tree. The root node is 33. The node 33 has the child nodes, 20 and 21. The node 20 has the child nodes, 34 and 87. The node 21 has the child nodes, 62 and 84. The node 34 has the child nodes, 3 and 17. The node 87 has the child nodes, 25 and 44. The node 62 has the child node, 10.

Back to Figure

A full binary tree. The root node is 33. The node 33 has the child nodes, 20 and 21. The node 20 has the child nodes, 34 and 87. The node 21 has the child nodes, 62 and 84. The node 34 has the child nodes, 3 and 17. The node 87 has the child nodes, 25 and 44. The node 62 has the child nodes, 10 and 70. The node 84 has the child nodes, 55 and 92.

Back to Figure

A height-unbalanced binary tree. The root node of this binary tree is 33. The node 33 has the child nodes, 20 and 21. The node 20 has the child nodes, 34 and 87. The node 21 has the child node, 84. The node 34 has the child node, 17. The node 87 has the child node, 25. The node 84 has the child node, 44.

Back to Figure

A diagram that explains how the attach Left and attach Right methods work. The root node is shows. The attach Left method takes the binary tree parameter named b t which may contain 0 or more elements and attaches its root to the left child of root. The attach Right method takes the binary tree parameter named b t which may contain 0 or more elements and attaches its root to the right child of root.

Back to Figure

A example of pre order, in order, and post order traversals in a tree that is performed by walking counterclockwise around the tree. The following binary tree is shown. The root node is 33. The node 33 has the children, 20 and 21. The node 20 has the children, 34 and 87. The node 21 has the child 84. The node 34 has the child, 17. The node 87 has the children, 25 and 44. The pre order, in order, and post order traversals in this tree happens in the following order. 33, 20, 34, 17, 20, 87, 25, 87, 44, 87, 20, 33, 21, and 84.

Back to Figure

A node is visited on three occasions. 1. Pre order. 2. In order. 3. Post order. 34, 17, and 34.

Back to Figure

The output of the program in Example 15.5. Pre order: 33, 20, 34, 17, 87, 25, 44, 21, 84. In order: 34, 17, 20, 25, 87, 44, 33, 21, 84. Post order: 17, 34, 25, 44, 87, 20, 84, 21, 33.

Back to Figure

A binary search tree in which values on the left are smaller than 50 and the value son the right are larger than 50. The root node in the binary tree is 50. It has the child nodes, 20 and 66. The node 20 has the child nodes, 5 and 36. The node 66 has the child node, 84. The node 5 has the child node, 17. The node 36 has the child node, 25 and 44.

Back to Figure

A binary tree that is not a binary search tree. The root node is 50. It has the child nodes, 20 and 66. The node 20 has the child nodes, 5 and 25. The node 66 has the child node, 84. The node 5 has the child node, 17. The node 25 has the child nodes, 36 and 44.

Back to Figure

The number 44 is searched in a binary search tree. The root node is 50. It has the child nodes, 20 and 66. The node 20 has the child nodes, 5 and 36. The node 66 has the child node, 84. The node 5 has the child node, 17. The node 36 has the child nodes, 25 and 44. The number 44 is located with the following traversals. 50, 20, 36, 44.

Back to Figure

A binary search tree that is similar to a linked list. The root node is 1. It has the child node, 2. The node 2 has the child node, 3. The node 3 has the child node, 4. The node 4 has the child node, 5.

Back to Figure

The number 60 is inserted in a binary search tree. The root node is 50. It has the child nodes, 20 and 66. The node 20 has the child nodes, 5 and 36. The node 66 has the child node, 84. The node 5 has the child node, 17. The node 36 has the child nodes, 25 and 44. The following traversals occur after which the number 60 is inserted. 50, 66.

Back to Figure

Insertions that are done in a binary search tree. The root node is 50. The node 20 is inserted as a child of the node 50. Node 36 is inserted as a child of node 20. Node 66 is inserted as a child of node 50. Node 25 is inserted as a child of node 36. Node 5 is inserted as a child of node 20. Node 17 is inserted as a child of node 5. Node 84 is inserted as a child of node 66. Node 44 is inserted as a child of node 36.

Back to Figure

The resulting binary search tree after the number 25 is deleted from it. The root node is 50. It has the child nodes, 20 and 66. The node 20 has the child nodes, 5 and 36. The node 66 has the child node, 84. The node 5 has the child node, 17. The node 36 has the child nodes, 44.

Back to Figure

The resulting binary search tree after the number 5 is deleted from it. The root node is 50. It has the child nodes, 20 and 66. The node 20 has the child nodes, 17 and 36. The node 66 has the child node, 84. The node 36 has the child nodes, 25 and 44.

Back to Figure

The two candidate replacement values when 50 is deleted from the binary search tree in Figure 15.13 The root node is 50. It has the child nodes, 20 and 66. The node 20 has the child nodes, 5 and 36. The node 66 has the child node, 84. The node 5 has the child node, 17. The node 36 has the child nodes, 44. The largest value on the left of 50 is 44, and the smallest value on the right of 50 is 66.

Back to Figure

The largest value in the left subtree is located. The root node is 50. It has the child nodes, 20 and 66. The node 20 has the child nodes, 5 and 36. The node 66 has the child node, 84. The node 5 has the child node, 17. The node 36 has the child nodes, 44. The following traversals are done to locate, 44. 50, 30, 36, 44.

Back to Figure

The smallest value in the right subtree is located. The root node is 50. It has the child nodes, 20 and 66. The node 20 has the child nodes, 5 and 36. The node 66 has the child node, 84. The node 5 has the child node, 17. The node 36 has the child nodes, 44. The following traversals are done to locate, 66. 50, 66.

Back to Figure

The resulting tree after 50 is deleted from it using the first scenario. The root node is 44. It has the child nodes, 20 and 66. The node 20 has the child nodes, 5 and 36. The node 66 has the child node, 84. The node 5 has the child node, 17. The node 36 has the child node, 25.

Back to Figure

The resulting tree after 50 is deleted from it using the second scenario. The root node is 66. It has the child nodes, 20 and 84. The node 20 has the child nodes, 5 and 36. The node 5 has the child node, 17. The node 36 has the child nodes, 25 and 44.

Back to Figure

The steps in deleting the number 50 stored in a node with two children. The instance n d represents the following binary search tree. The root node is 50. It has the child nodes, 20 and 66. The node 20 has the child nodes, 5 and 36. The node 66 has the child node, 84. The node 5 has the child node, 17. The node 36 has the child nodes, 25 and 44. 44 is the largest value on the left side. 50 is changed to 44. 44 is deleted in the subtree whose root contains 20.

Back to Figure

The number 5 is deleted from the binary search tree. The binary search tree has the following nodes. The root node is 50. It has the child nodes, 20 and 66. The node 20 has the child nodes, 5 and 36. The node 66 has the child node, 84. The node 5 has the child node, 17. The node 36 has the child nodes, 25 and 44. 5 is deleted. An edge points from 50 to 20. The edge from 20 is set to 17. The links between 20 to 5 and 5 to 17 are reset when 5 is deleted.

Back to Figure

The output of the program in Example 15.7. Pre order: 50, 20, 5, 17, 36, 25, 44, 66, 84. In order: 5, 17, 20, 25, 36, 44, 50, 66, 84. Post order: 17, 5, 25, 44, 36, 20, 84, 66, 50. Deleting 50. Pre order: 44, 20, 5, 17, 36, 25, 66, 84. In order: 5, 17, 20, 25, 36, 44, 66, 84. Post order: 17, 5, 25, 36, 20, 84, 66, 44. Deleting 8. Pre order: In order: Post order:

Back to Figure

The resulting array when the binary search tree is unloaded by traversing in order. The index positions and values of the resulting array are listed further. 0, 5; 1, 17; 2, 20; 3, 25; 4, 36; 5, 44; 6, 50; 7, 66; 8, 84.

Back to Figure

The balanced binary search tree that is built from the array. The root node is 36. It has the child nodes, 17 and 50. The node 17 has the child node, 5 and 20. The node 50 has the child nodes, 44 and 66. The node 66 has the child node, 84.

Back to Figure

The output of the program in Example 15.8. Pre order: 5, 17, 20, 25, 36, 44, 50, 66, 84. In order: 5, 17, 20, 25, 36, 44, 50, 66, 84. Post order: 84, 66, 50, 44, 36, 25, 20, 20, 17, 5. After rebuilding the tree. Pre order: 36, 17, 5, 20, 25, 50, 44, 66, 84. In order: 5, 17, 20, 25, 36, 44, 50, 66, 84. Post order: 5, 25, 20, 17, 44, 84, 66, 50, 36.

Back to Figure

The inheritance hierarchy for J Frame is as follows. Object. Component. Container. Window. Frame. J Frame.

Back to Figure

A window that is generated by the code in Example 16.1. The window has the following text in the title bar. A Shell G U I Application. The window has minimize, maximize, and close buttons.

Back to Figure

The inheritance hierarchy for some G U I classes. J Radio Button and J Check Box inherit from J Toggle Button. J Button and J Toggle Button inherit from Abstract Button.

Abstract Button, J List and J Combo Box inherit from J Component. J Password Field inherits from J Text Field. J Text Field and J Text Area inherit from J Text Component. J Label and J Text Component inherit from J Component. J Component inherits from Container. Container inherits from Component. Component inherits from Object. Object is the class that is at the top of the hierarchy.

Back to Figure

The window that is displayed on running Example 16.2. A window is displayed with the following text in the title bar. “What's for dinner?” Inside the window is the following label. Sushi tonight? A photograph of sushi is displayed to the right of the label. A tooltip is displayed with the following text. photo of sushi.

Back to Figure

Hierarchy of the Event class. Mouse Event inherits from Input Event. Input Event inherits from Component Event. Action Event, Item Event, and Component Event inherit from A W T Event. List Selection Event and A W T Event inherit from Event Object. Event Object inherits from Object.

Back to Figure

Running the program in Example 16.3. A window is displayed with the following text in the title bar. Login Screen. It has two input fields with the labels, Enter I d and Enter password. A warning message is displayed below the input boxes stating that illegal login will be punishable by law. In this window, the i d is entered as wrong I D and four question marks are entered as the password. In one more attempt, open is entered as the i d and six question marks are entered as the password.

Back to Figure

Running the program in Example 16.7. A window is displayed with the text, “Simple Math” in the title bar. An input field is present in the window with the label, Enter a number. There are two buttons below the input field with the labels, Square and Cube. The Square button has been clicked. Below the buttons, the result is displayed as 16.

Back to Figure

The program in Example 16.5 is executed. It displays a window with the following text in the title bar. Selecting a color. There are three radio buttons with the labels red, green, and blue. Below, the radio buttons is the following text. Watch my background. The background of this text changes according to the radio button that is selected.

Back to Figure

The program in Example 16.6 is executed. The program displays a window with the following text in the title bar. Selecting a color. Three checkboxes are displayed in the window with the labels, red, green, and blue. The red and green checkboxes are checked. Below the checkboxes, the following text is displayed with a background color. Watch my background.

Back to Figure

A G U I window with a circuit displayed in it. The window shows an electrical circuit with a power source, light bulb, and a switch. Two buttons are present above the circuit, labeled Open and Close. The Open button has been clicked. The switch in the circuit is displayed in the open position.

Back to Figure

A G U I window with a circuit displayed in it. The window shows an electrical circuit with a power source, light bulb, and a switch. Two buttons are present above the circuit, labeled Open and Close. The Close button has been clicked. The switch in the circuit is displayed in the close position. The light bulb is displayed in yellow, since it is alight.

Back to Figure

The window that is displayed when the program in Example 16.8 is executed. The window displays the following text in the title bar. Food samplings of various countries. A J Combo Box is present in the window. The selection in the J Combo Box is Greece. Next to the combo box is an image.

Back to Figure

U M L diagram for the classes Sub Hunt and Sub Hunt View Controller. Sub Hunt has a composition relationship with Sub Hunt View Controller. Mouse Handler has a composition relationship with Sub Hunt View Controller. Mouse Handler inherits from Mouse Adapter. Sub Hunt has the following data members and methods. The data members are the following. DEFAULT underscore GAME underscore SIZE with i n t data type. SIDE with i n t data type. game Size with i n t data type. x C t r with i n t data type. y C t r with i n t data type. status with String data type. hit with boolean data type. The methods are the following. Sub Hunt with i n t parameter. get Status with String return type. Get Game Size with i n t return type. is Hit with boolean return type. play with two i n t parameters and void return type. draw with Graphics, and two i n t parameters and void return type. Sub Hunt View Controller has the following data members and methods. The data members are the following. sub with Sub Hunt data type. x with i n t data type. y with i n t data type. game Started with boolean data type. insets with Insets data type. m h with Mouse Handler data type. The methods are the following. Sub Hunt View Controller with Sub Hunt parameter. paint with Graphics parameter and void return type. Mouse Handler has the following method. mouse Clicked with Mouse Event parameter and void return type. The Mouse Adapter class is present in the package java dot a w t dot event.

Back to Figure

A J Frame is shown in gray with the sub hunt game rectangle inside it in orange. The top, right, bottom, and left boundaries between the rectangle and the J Frame are indicated.

Back to Figure

The sub hunt game program executing. A window is displayed with the following text in the title bar. Sunk! Many blue dots are displayed on the screen.

Back to Figure

U M L diagram for the Treasure Hunt game. The following details are shown in the diagram. Treasure Hunt has Treasure Hunt View Controller. Treasure Hunt View Controller implements Mouse Motion Listener in the packet java, dot, a w t, dot event package. Treasure Hunt class has the following data members. DEFAULT underscore GAME underscore SIZE constant of i n t data type. SIDE constant of i n t data type. game Size of i n t data type. x C t r of i n t data type. y C t r of i n t data type. status of String data type. game Over of boolean data type. The following methods are also present. Treasure Hunt with i n t parameter. get Status with String return type. get Game Size with i n t return type. is Game Over with boolean return type. play with two i n t parameters and void return type. draw with a Graphics and two i n t return types. It has a void return type. Treasure Hunt View Controller class has the following data members. hunt of Treasure Hunt data type. x of i n t data type. y of i n t data type. insets of Insets data type. The following methods are also present. Treasure Hunt View Controller with Treasure Hunt parameter. mouse Moved with Mouse Event parameter and void return type. mouse Dragged with Mouse Event parameter and void return type. paint with Graphics parameter and void return type.

Back to Figure

The Play Treasure Hunt program executing. A window is displayed with the word, “Play!” displayed in the title bar. The word, warm is displayed in the window with the mouse pointer on it. The user is getting close to the treasure.

Back to Figure

The Play Treasure Hunt program executing. A window is displayed with the word, “Play!” displayed in the title bar. The user has found the treasure. Inside the window, a red box is displayed with three dollar symbols under the mouse pointer indicating that the treasure has been found.

Back to Figure

The program in example 16.15 runs and displays a window with a chess board in white and red grids. In the title bar, the following text is displayed. Click a square to display its position. The following grid positions are displayed. A 8. b 2. b 5. c 3. d 7. e 4. f 2. f 8. h 1.

Back to Figure

A window displaying the winning position of a 3-by-3 tile puzzle game. The window displays tiles in a 3 by 3 grid. The tiles are numbered from 1 to 8 from top to bottom and left to right. The last tile in the bottom row is blank.

Back to Figure

U M L Diagram for Tile Puzzle and Tile Puzzle View Controller. The following details are shown in the diagram. Tile Puzzle has Tile Puzzle View Controller. Grid Button has Tile Puzzle View Controller. Button Handler is nested in Tile Puzzle View Controller. Grid Button implements J Button. Tile Puzzle class has the following data members. tiles with data type of String array. side with i n t data type. empty Row with i n t data type. empty C o l with i n t data type. The following methods are also present. Tile Puzzle with i n t parameter. set Up Game with void return type. get Side with i n t return type. get Tiles with String array return type. try To Play with two i n t parameters and boolean return type. possible To Play with two i n t parameters and boolean return type. won with boolean return type. Tile Puzzle View Controller has the following data members. game with Tile Puzzle data type. contents with Container data type. squares with Grid Button array data type. The following methods are also present. Tile Puzzle View Controller with Tile Puzzle data type. set Up Game G U I with void return type. Up data with void return type. Grid Button class has the following data members. row with i n t data type. column with i n t data type. The following methods are present. Grid Button with a String and two i n t parameters. get Row with i n t return type. get Column with i n t return type. set Row with i n t parameter and void return type. set Column with i n t parameter and void return type. Button Handler class has the following method. action Performed with Action Event parameter and void return type. J Button is a class in the java x dot swing package.

Back to Figure

A window displays tile puzzle game. In the title bar, the following text is displayed. The Tile Puzzle Game. The window displays tiles in a 3 by 3 grid. The first row of tiles are numbered 1 to 3. The second row of tiles are numbered 8, 5, and blank. The third row of tiles are numbered 8, 7, and 4.

Back to Figure

A rectangular box displays the areas in Border Layout. The areas are North, West, Center, East, and South representing geographical directions in the box.

Back to Figure

The U M L Diagram for Classes of an application. The following details are shown in the diagram. Border Layout View has R G B Color Mixer Controller. R G B Color Mixer Controller implements Action Listener. One more class is R G B Color Mixer. Border Layout View class has the following data members. center with J Label data type. north with J Button data type. east with J Button data type. south with J Button data type. west with J Button data type. The following methods are also present. Border Layout View. get North with J Button return type. get East with J Button return type. get South with J Button return type. get West with J Button return type. get Center with J Label return type. R G B Color Mixer Controller class has the following data members. model with R G B Color Mixer data type. view with Border Layout View data type. The following methods are also present. R G B Color Mixer View with R G B Color Mixer and Border Layout View parameters. action Performed with Action Event parameter and void return type. update View with void return type. R G B Color Mixer has the following data members. POSITION underscore CHANGE constant with i n t data type. POSITION underscore RED constant with i n t data type. POSITION underscore GREEN constant with i n t data type. POSITION underscore GREEN constant with i n t return type. POSITION underscore BLUE constant with i n t data type. delta with i n t data type. color with Color data type. The following methods are also present. R G B Color Mixer with i n t parameter. get Color with Color return type. invert Delta with void return type. add To Red with void return type. add To Green with void return type. add To Blue with void return type. get Method Label with i n t parameter and String return type. get Title with String return type. to String with String return type.

Back to Figure

The Mix Colors application executing. A window is displayed with the text, “Start Mixing Colors” displayed in the title bar. Inside the window, in the north direction is a button with the label, “CHANGE DIRECTION.” In the east, west, and south directions are the buttons with the labels, RED, BLUE, and GREEN. In the center, a color shade is displayed with the following label. r equals 175. g equals 135. b equals 155.

Back to Figure

A window that is displayed when the Color Frequency game executes. The window has the following text displayed in the title bar. What is the most frequent color? Inside the window, in the west side, there are three buttons laid out in a single column with the labels, RED, GREEN, and BLUE. The rest of the window features several squares colored, red, blue, and green.

Back to Figure

The underlying nested layout for the display in a program. The window is divided into the west and center regions. In the west region is a J Panel managed by a 3 by 1 Grid Layout. In the center is a J Panel managed by an 8 by 8 Grid Layout.

Back to Figure

The U M L diagram for the various classes and interface in this application. The diagram shows the following information. Color Grid Game Controller has Bordered Label Grid View. Color Grid Game Controller has Color Grid Game. Color Grid Game Controller implements Action Listener. Color Frequency Game implements Color Grid Game. The interface Action Listener is from the package java dot a w t dot event. Color Grid Game Controller class has the following data members. model with Color Grid Game data type. View with Bordered Label Grid View data type. The following methods are also present. Color Grid Game Controller with Color Grid Game and Bordered label Grid View parameters. set Up Game with void return type. set Up Labels with void return type. set Up Buttons with void return type. action Performed with Action Event data type and void return type. Bordered Label Grid View class has the following data members. button Panel with J Panel data type. buttons with J Button array data type. grid with J Panel data type. labels with J Label two dimensional array data type. The following methods are also present. Bordered Label Grid View with two i n t parameters. set Button Text with an i n t and String parameter with void return type. set Button Listener with i n t and Action Listener parameters and void return type. set Label Background with two i n t and Color parameters with void return type. Color Grid Game interface has the following methods. get Title with String return type. get Size with i n t return type. get Number Of Colors with i n t return type. get Label with i n t parameter and String return type. get Index with String parameter and i n t return type. is Correct with i n t parameter and boolean return type. get Grid Color with two i n t parameters and Color return type. Color Frequency Game class has the following data members. DEFAULT underscore SIZE constant with i n t data type. grid Colors with Color two dimensional array data type. coloring Colors with Color data type. counts with int array data type. labels with String array data type. The following methods are also present. Color Frequency Game with i n t, Color array, and String array parameters. fill Grid With Colors with void return type. highest Count with i n t return type. get Title with String return type. get Size with i n t return type. get Number Of Colors with i n t return type. get Label with i n t return type. get Label with i n t parameter and String return type. is Correct with i n t parameter and boolean return type. get Grid Color with two i n t parameters and Color return type.

Back to Figure

In the Color Frequency game, the user has to find the more frequent color in a grid of colors. Red, Green, and Blue buttons are given for the user to select an answer. The user clicks on the button BLUE and the program displays a message box with the following text and an O K button. You won.

Back to Figure

The Tile Puzzle game executing. A window is displayed with the following text in the title bar. Practicing layout managers. The following label is displayed at the bottom of the window. Have fun playing this Tile Puzzle game.

Back to Figure

The Tile Puzzle game executing. A window is displayed with the following text in the title bar. Practicing layout managers. The following label is displayed at the bottom of the window. Have fun playing this Tile Puzzle game. A row of buttons is present for the user to make a selection for the number of tiles. The buttons have the following labels. 3 by 3. 4 by 4. 5 by 5. The user has clicked on the 3 by 3 button. 3 rows of 3 tiles are displayed in a grid numbered from 1 to 8 randomly along with one blank tile.

Back to Figure

The root Node is labeled Object. Node class inherits from the Object Class. Parent class inherits from the Node class. Region class inherits from the Parent class. Control class inherits from the Region class. The 4 classes inheriting from the Control class are as follows: Text Input Control, Labeled, Slider, Combo Box Base, left angled bracket, uppercase t, right angled bracket. The Text Field class inherits from the Text Input Control class. The Button Base and Label classes inherit from the Labeled class. The Combo Box, left angled bracket, uppercase t, right angled bracket, class inherits from the Combo Box Base, left angled bracket, uppercase t, right angled bracket, class. The Text Field Class inherits from the Text Input Control class. The classes which inherit from the Labeled class are as follows: Button Base and Label. The class which inherits from the Combo Box Base, left angled bracket, uppercase t, right angled bracket, class is, Combo Box, left angled bracket, uppercase t, right angled bracket. The classes which inherit from the Button Base class are as follows: Button, Check Box, Toggle Button. The class which inherits from the Toggle Button class is Radio Button.

Back to Figure

The root Node is labeled Object. Node class inherits from the Object Class. Parent class inherits from the Node class. Region class inherits from the Parent class. Pane class inherits from the Region class. The classes inheriting from the Pane class are as follows: Border Pane, Grid Pane, H Box, Stack Pane, and V Box.

Back to Figure

The window has a caption which reads: Sushi tonight, question mark. A photo below the caption displays a plate of Sushi.

Back to Figure

The window has a text label, a text box, 2 buttons and another text label below the second button. The first text label reads: Enter a number. The cursor is within the text box. The buttons are labeled Square and Cube, respectively. The second text label reads: Result.

Back to Figure

The window has a text label, a text box, 2 buttons and another text label below the second button. The first text label reads: Enter a number. The entry within the text box reads: 3. The buttons are labeled Square and Cube, respectively. The button labeled cube is selected. The second text label reads: Result. The text displayed at the bottom of the window reads: 27.0.

Back to Figure

The window has 3 radio buttons and a caption with a red background. The radio buttons are labeled red, green, and blue. The radio button labeled red is selected. The caption reads: Watch my background.

Back to Figure

The window has 3 check boxes and a text label. The check boxes are labeled red, green and blue. The text label reads: Watch my background.

Back to Figure

The window has 2 buttons and a circuit diagram. The buttons are labeled OPEN and CLOSE. The circuit diagram has a Light bulb and an open switch in series with a Power source. The button labeled OPEN is selected.

Back to Figure

The window has 2 buttons and a circuit diagram. The buttons are labeled OPEN and CLOSE. The circuit diagram has a Light bulb and a closed switch in series with a Power source. The button labeled CLOSE is selected. The light bulb glows.

Back to Figure

The window has a drop down selection box and an image. The selected option is labeled France. The image displays a plate containing French cuisine.

Back to Figure

The window displays a color image and a grayscale image. A scale at the bottom of the window ranges from 0 to 0.33. The slider on the scale is set to 0.165.

Back to Figure

The window displays a chessboard. The white squares in the first row and the first, fifth, and seventh columns are marked a 8, e 8, and g 8, respectively. The red squares in the first row and the sixth and eighth columns are marked f 8 and h 8, respectively. The white square in the second row and second column is labeled b 7. The red square in the second row and third column is marked c 7. The red square in the third row and fourth column is marked d 6. The white square in the fourth row and sixth column is labeled f 5. The white square in the sixth row and second column is marked b 3. The red squares in the eighth row and the first and third columns are marked a 1 and c 1, respectively. The white squares in the eighth row and the second and eighth columns are marked b 1 and h 1, respectively.

Back to Figure

The window displays a chessboard. The white square in the first row and first column is marked a 8. The white square in the third row and fifth column is marked e 6. The red square in the sixth row and third column is marked c 3. The red square in the seventh row and eighth column is marked h 2. The white square in the eighth row and eighth column is marked h 1.

Back to Figure

The window has tiles arranged in 3 rows and 3 columns. The tiles in the first row are marked 1, 2, and 3. The tiles in the second row are marked 4, 5, and 6. The tiles in the third row are marked 7, 8, and blank.

Back to Figure

The classes shown in the diagram are titled Tile Puzzle, Grid Button, Tile Puzzle View Controller, and Button Handler. The attributes of the Tile Puzzle class are as follows: tiles, colon, Strings, left square bracket, right square bracket, left square bracket, right square bracket, side, colon, int, empty Row, colon, int, empty Col, colon, int. The methods of the Tile Puzzle class are as follows: Tile Puzzle, left parentheses, right parentheses, set Up Game, left parentheses, right parentheses, colon, void, get Side, left parentheses, right parentheses, colon, int, get Tiles, left parentheses, right parentheses, colon, String, left square bracket, right square bracket, left square bracket, right square bracket, try To Play, left parentheses, int, comma, int, right parentheses, colon, boolean, possible To Play, left parentheses, int, comma, int, right parentheses, colon, boolean, won, left parentheses, right parentheses, colon, boolean. The attributes of the Grid Button class are as follows: row, colon, int, column, colon, int. The methods of the Grid Button class are as follows: Grid Button, left parentheses, String, comma, int, comma, int, right parentheses, get Row, left parentheses, right parentheses, colon, int, get Column, left parentheses, right parentheses, colon, int, set Row, left parentheses, int, right parentheses, colon, void, set Column, left parentheses, int, right parentheses, colon, void. The attributes of the Tile Puzzle View Controller class are are follows: game, colon, Tile Puzzle, squares, colon, Grid Button, left square bracket, right square bracket, left square bracket, right square bracket.

The methods of the Tile Puzzle View Controller Class are as follows: Tile Puzzle View Controller, left parentheses, Tile Puzzle, right parentheses, set Up Game G U I, left parentheses, right parentheses, colon, void, update, left parentheses, right parentheses, colon, void, show Message Dialog, left parentheses, String, comma, String, right parentheses, colon, void. Lines with a shaded diamond at the end are drawn from the Tile Puzzle and the Grid Button classes to the Tile Puzzle View Controller class. The method in the Button Hander class is handle, left parentheses, Action Event, right parentheses, colon, void. A line terminating in a circle with a plus symbol within it is drawn from the Button Handler class to the Tile Puzzle View Controller class. An arrow is drawn from the Grid Button class to Button which is in the java f x dot scene dot control package.

Back to Figure

The window displays 9 tiles arranged in 3 rows and 3 columns. The labels of the tiles in the first row are 1, 2, and 3. The labels of the tiles in the second row are 6, 5, and blank. The labels of the tiles in the third row are 8, 7, and 4.

Back to Figure

A large rectangle is subdivided into 3 smaller rectangles from top to bottom. The first and third rectangles are labeled top and bottom, respectively. The second rectangle is subdivided into 3 smaller rectangles from left to right and they are labeled, left, center, and right, respectively.

Back to Figure

The window has 2 panes. The first pane has 6 buttons labeled Fade, Rotate, Scale, Combine Animations, Start Translate, and Stop Translate. The last button is deactivated. The second pane displays the image of a sprite.

Back to Figure

The Color Frequency Game Figure A shows the Color Frequency game executing. A window is displayed with the following text in the title bar. What is the most frequent color? In the left side of the window are three buttons with the labels RED, GREEN, and BLUE laid out in a single column. Next to the buttons is a color grid of with red, green, and blue squares.

Back to Figure

The underlying nested layout used in the Color Frequency game. Figure B shows the underlying nested layout in the Color Frequency game. In the LEFT side of the window is a V Box with 3 buttons. In the CENTER is a Grid Pane with 64 buttons organized in 8 rows and 8 columns.

Back to Figure

The U M L diagram for the Color Frequency Game application. The diagram shows the following details. Color Grid Game Controller has Bordered Label Grid View. Color Grid Game Controller has Color Grid Game. Color Frequency Game implements Color Grid Game. Color Grid Game Controller implements Event Handler. Color Grid Game Controller has the following data members. model with Color Grid Game data type. view with Bordered Label Grid View data type. This class has the following methods. Color Grid Game Controller with Color Grid Game and Bordered Label Grid View parameters. set Up Game with void return type. set Up Labels with void return type. set Up Buttons with void return type. handle with Action Event parameter and void return type. Bordered Label Grid View has the following data members. button Panel with V Box data type. buttons with Button array data type. grid with Grid Pane data type. labels with Label two dimensional array. This class has the following methods. Bordered Label Grid View with two i n t parameters. set Button Text with i n t and String parameters and void return type. set Button Listener with i n t and Event Handler, left angle bracket, Action Event, right angle bracket parameters and void return type. set Label Background with two i n t and one String parameters with void return type. Color Grid Game interface has the following methods. get Title with String return type. get Size with i n t return type. get Number Of Colors with i n t return type. get Label with i n t parameter and String return type. get Index with String parameter and i n t return type. is Correct with i n t parameter and boolean return type. get Grid Hex Color with two i n t parameters and String return type. Color Frequency Game class has the following data members. DEFAULT underscore SIZE constant with i n t data type. grid Colors with Color two dimensional array data type. coloring Colors with Color array data type. counts with i n t array data type. labels with String array data type. This class has the following methods. Color Frequency Game with i n t, Color array, and String array parameters. fill Grid With Colors with void return type. get Size with i n t return type. get Number Of Colors with i n t return type. get Number Of Colors with i n t return type. get Grid Hex Color with two i n t parameters and String return type. to Hex with i n t parameter and String return type. highest Count with i n t return type. is Correct with i n t parameter and boolean return type. get Title with String return type. get Label with i n t parameter and String return type. get Index with String parameter and i n t return type. Interface Event Handler, left angle bracket, Action Event, right angle bracket is present in the java f x dot event package. Figure 17.2 Chapter 17 In the Color Frequency game, the user has to find the more frequent color in a grid of colors. Red, Green, and Blue buttons are given for the user to select an answer. The user clicks on the button GREEN and the program displays a message box with the following text and an O K button. You won.

Back to Figure

U M L diagram for the Sub Hunt game. The following details are shown in the diagram. Sub Hunt has Sub Hunt View Controller. Sub Hunt class has the following data members. DEFAULT underscore GAME underscore SIZE constant with i n t data type. SIDE constant with i n t data type. game Size with i n t data type. x C t r with i n t data type. y C t r with data type. status with String data type. h i t with boolean data type. The following methods are also present. Sub Hunt with i n t parameter. get Game Size with i n t return type. get Status with i n t return type. is Hit with boolean return type. play with two i n t parameters and void return type. draw with Graphics Context and two i n t parameters, and void return type. Sub Hunt View Controller class has the following data members. hunt with Sub Hunt data type. stage with Stage data type. g c with Graphics Context data type. The following methods are also present. Sub Hunt View Controller with Sub Hunt, Stage, and two i n t parameters. handle Mouse Event with Mouse Event parameter and void return type. handle Touch Event with Touch Event parameter and void return type. play with two i n t parameters and void return type.

Back to Figure

The Sub Hunt game running in a window. A submarine is indicated by a black colored graphic and there are other blue dots in the window.

Back to Figure

U M L diagram for the Sub Hunt game. The following details are shown in the diagram. Treasure Hunt has Treasure Hunt View Controller. Treasure Hunt View Controller implements Event Handler. The Treasure Hunt class has the following data members. DEFAULT underscore GAME underscore SIZE constant with i n t data type. SIDE constant with i n t data type. game Size with i n t data type. x C t r with i n t data type. y C t r with i n t data type. status with String data type. game Over with boolean data type. This class has the following methods as well. Treasure Hunt with i n t parameter. get Game Size with i n t return type. is Game Over with boolean return type. play with two i n t parameters and void return type. draw with a Graphics and two i n t parameters and void return type. The Treasure Hunt View Controller class has the following data members. width with i n t data type. height with i n t data type. hunt with Treasure Hunt data type. g c with Graphics Context data type. This class has the following methods as well. Treasure Hunt View Controller with Treasure Hunt and two i n t parameters. handle with Input Event parameter and void return type.

Back to Figure

The user getting close to the treasure in the Play Treasure Hunt game. A window is displayed with the word, “Play!” displayed in the title bar. The word, warm is displayed in the window with the mouse pointer on it.

Back to Figure

The user finding the treasure in the Play Treasure Hunt game. A window is present with the word, “Thank you for playing” displayed in the title bar. Inside the window, a red box is displayed with three dollar symbols under the mouse pointer indicating that the treasure has been found.

Back to Figure

A program that displays a pie chart for monthly expenses. A window with a pie chart is shown. In the title bar, the following text is displayed. Pie Chart for Monthly Expenses. The pie chart shows the following details. Rent: 50 percent. Food: 27 percent. Phone: 5 percent. Entertainment: 11 percent. Utilities: 7 percent.

Back to Figure

A bar chart for stocks and bonds displayed using the first set of values for stocks. The chart is labeled, “Next Scenario.” The following details are indicated in the bar chart. 2020: Bonds: 100; Stocks, 100. 2021: Bonds: 102; Stocks, 108. 2022: Bonds: 106; Stocks, 116. 2023: Bonds: 108; Stocks, 122.

Back to Figure

A bar chart for stocks and bonds displayed using the second set of values for stocks. The chart is labeled, “Next Scenario.” The following details are indicated in the bar chart. 2020: Bonds: 100; Stocks, 98. 2021: Bonds:

104; Stocks, 100. 2022: Bonds: 106; Stocks, 102. 2023: Bonds: 108; Stocks, 104.

Back to Figure

The Simple Math application running. A window is displayed with the text “ Simple Math” listed in the title bar. A label is present with the text, “Enter a number.” A text field is present for input. Below the text field are two buttons. One for Square and the other for Cube. Below these buttons is a label with the text, “Result.” The Cube button changes color to green when the mouse hovers over it.

Back to Figure

The Simple Math application running. A window is displayed with the text “ Simple Math” listed in the title bar. A label is present with the text, “Enter a number.” A text field is present for input. Below the text field are two buttons. One for Square and the other for Cube. Below these buttons is a label with the text, “Result.” The number 3 is entered in the text field and the Cube button is clicked. This prints the number 27.0 below the result label.

Back to Figure

A window displayed for the Tile Puzzle Game. A window is shown with the text “Practice Nested Layouts” displayed in the title bar. The window is empty. In the bottom left corner of the window, a label is displayed with the text “Have fun playing this Tile Puzzle game.”

Back to Figure

A window displayed for the Tile Puzzle Game. A window is shown with the text “Practice Nested Layouts” displayed in the title bar. Inside the window there are three labels displayed in the top row with the following text. 3 by 3. 4 by 4. 5 by 5. Below this row there are three rows and three columns of square buttons with numbers displayed in them. In the first row are the numbers 8, 7, and 6. In the second row are the numbers 5, 4, and 3. In the third row are the numbers 2, 1, and blank.

Back to Figure

The table has 2 columns labelled Memory Unit and Size. The row entries are as follows.
Row 1. Memory Unit: K B or K bytes or kilobytes. Size: About 1000 bytes, left parentheses, exactly 2 to the power of 10 or 1024 bytes, right parentheses.
Row 2. Memory Unit: M B or M bytes or megabytes. Size: About 1 million bytes, left parentheses, exactly 2 to the power of 20 or 1,048,576 bytes, right parentheses.
Row 3. Memory Unit: G B or G bytes or gigabytes. Size: About 1 billion bytes, left parentheses, exactly 2 to the power of 30 or 1,073,741,824 bytes, right parentheses.
Row 4. Memory Unit: T B or T bytes or terabytes. Size: About 1 trillion bytes, left parentheses, exactly 2 to the

power of 40 or 1.09951 times 10 to the power of 21 bytes, right parentheses.

Back to Table

The table has 6 columns labelled Device, Location, Type, Speed, Capacity in M B, and cost per K B. The row entries are as follows.
Row 1. Device: L 1 cache. Location: on-chip. Type: S R A M. Speed: Very fast. Capacity in M B: Very small. Cost per K B: Very high.
Row 2. Device: L 2 cache. Location: off-chip. Type: S R A M. Speed: Fast. Capacity in M B: Small. Cost per K B: High.
Row 3. Device: Memory. Location: off-chip. Type: D R A M. Speed: Moderate. Capacity in M B: Moderate. Cost per K B: Moderate.
Row 4. Device: Hard disk. Location: Separate. Type: Disk media. Speed: Slow. Capacity in M B: Large. Cost per K B: Small.

Back to Table

The equation reads: 359, equals, 3 times 10 to the power of 2, plus, 5 times 10 to the power of 1, plus, 9 times 10 to the power of 0.

Back to Code

The equation reads: 11011, equals, 1 times 2 to the power of 4, plus, 1 times 2 to the power of 3, plus, 0 times

2 to the power of 2, plus, 1 times 2 to the power of 1, plus, 1 times 2 to the power of 0.

Back to Code

The table has 2 columns labelled Decimal and Binary.
The row entries are as follows.
Row 1. Decimal: 0. Binary: 0 0 0 0.
Row 2. Decimal: 1. Binary: 0 0 0 1.
Row 3. Decimal: 2. Binary: 0 0 1 0.
Row 4. Decimal: 3. Binary: 0 0 1 1.
Row 5. Decimal: 4. Binary: 0 1 0 0.
Row 6. Decimal: 5. Binary: 0 1 0 1.
Row 7. Decimal: 6. Binary: 0 1 1 0.
Row 8. Decimal: 7. Binary: 0 1 1 1.
Row 9. Decimal: 8. Binary: 1 0 0 0.

Back to Table

The column entries from left to right are as follows.
Column 1. 2 to the power of 14: 16,384.
Column 2. 2 to the power of 13: 8,192.
Column 3. 2 to the power of 12: 4,096.
Column 4. 2 to the power of 11: 2,048.
Column 5. 2 to the power of 10: 1,024.
Column 6. 2 to the power of 9: 512.
Column 7. 2 to the power of 8: 256.
Column 8. 2 to the power of 7: 128.
Column 9. 2 to the power of 6: 64.
Column 10. 2 to the power of 5: 32.

Column 11. 2 to the power of 4: 16.
Column 12. 2 to the power of 3: 8.
Column 13. 2 to the power of 2: 4.
Column 14. 2 to the power of 1: 2.
Column 15. 2 to the power of 0: 1.

Back to Table

The equation reads: 11010, equals, 1 times 2 to the power of 4, plus, 1 times 2 to the power of 3, plus, 0 times 2 to the power of 2, plus, 1 times 2 to the power of 1, plus, 0 times 2 to the power of 0. This equals 16, plus, 8, plus, 0, plus, 2, plus 0 and this evaluates to 26.

Back to Code

The equation reads: 359, equals, 2 to the power of 8 times 1, plus, 103.

Back to Code

The equation reads: 359, equals, 2 to the power of 8 times 1, plus, 2 to the power of 7 times 0, plus, 2 to the power of 6 times 1, plus, 39.

Back to Code

The equation reads: 359, equals, 2 to the power of 8 times 1, plus, 2 to the power of 7 times 0, plus, 2 to the power of 6 times 1, plus, 2 to the power of 5 times 1, plus, 7.

Back to Code

The equation reads: 359, equals, 2 to the power of 8 times 1, plus, 2 to the power of 7 times 0, plus, 2 to the power of 6 times 1, plus, 2 to the power of 5 times 1, plus, 2 to the power of 4 times 0, plus, 2 to the power of 3 times 0, plus, 2 to the power of 2 times 1, plus, 3.

Back to Code

The equation reads: 359, equals, 2 to the power of 8 times 1, plus, 2 to the power of 7 times 0, plus, 2 to the power of 6 times 1, plus, 2 to the power of 5 times 1, plus, 2 to the power of 4 times 0, plus, 2 to the power of 3 times 0, plus, 2 to the power of 2 times 1, plus, 2 to the power of 1 times 1, plus, 1.

Back to Code

The equation reads: 359, equals, 2 to the power of 8 times 1, plus, 2 to the power of 7 times 0, plus, 2 to the power of 6 times 1, plus, 2 to the power of 5 times 1, plus, 2 to the power of 4 times 0, plus, 2 to the power of 3 times 0, plus, 2 to the power of 2 times 1, plus, 2 to the power of 1 times 1, plus, 2 to the power of 0 times 1.

Back to Code

The equation reads: 359, equals, 2 to the power of 8 times 1, plus, 2 to the power of 7 times 0, plus, 2 to the power of 6 times 1, plus, 2 to the power of 5 times 1, plus, 2 to the power of 4 times 0, plus, 2 to the power of 3 times 0, plus, 2 to the power of 2 times 1, plus, 2 to the power of 1 times 1, plus, 2 to the power of 0 times 1. This equals 1 0 1 1 0 0 1 1 1.

Back to Code

The table has 2 columns labelled Hex Digit and Binary Value. The row entries are as follows.
Row 1. Hex Digit: 0. Binary Value: 0 0 0 0.
Row 2. Hex Digit: 1. Binary Value: 0 0 0 1.
Row 3. Hex Digit: 2. Binary Value: 0 0 1 0.
Row 4. Hex Digit: 3. Binary Value: 0 0 1 1.
Row 5. Hex Digit: 4. Binary Value: 0 1 0 0.
Row 6. Hex Digit: 5. Binary Value: 0 1 0 1.
Row 7. Hex Digit: 6. Binary Value: 0 1 1 0.
Row 8. Hex Digit: 7. Binary Value: 0 1 1 1.
Row 9. Hex Digit: 8. Binary Value: 1 0 0 0.
Row 10. Hex Digit: 9. Binary Value: 1 0 0 1.
Row 11. Hex Digit: A. Binary Value: 1 0 1 0.
Row 12. Hex Digit: B. Binary Value: 1 0 1 1.
Row 13. Hex Digit: C. Binary Value: 1 1 0 0.
Row 14. Hex Digit: D. Binary Value: 1 1 0 1.
Row 15. Hex Digit: E. Binary Value: 1 1 1 0.
Row 16. Hex Digit: F. Binary Value: 1 1 1 1.

Back to Table

The bits of the binary number are grouped in sets of 4. The binary number from the most significant to the least significant bit is as follows: 0 0 0 1, blank space, 1 0 1 0, blank space, 1 1 1 1, blank space, 1 0 0 1, blank space, 1 0 1 1, blank space, 0 0 1 1, blank space, 1 0 1 1, blank space, 1 1 1 0. The hexadecimal equivalent of the binary number is as follows: 1 A F 9 B 3 B E.

Back to Code

The bits of the binary number are grouped in sets of 4. The binary number from the most significant to the least significant bit is as follows: 1 1 0 0, blank space, 1 0 1 0, blank space, 1 1 1 1, blank space, 1 1 1 0, blank space, 1 0 1 1, blank space, 1 0 1 0, blank space, 1 0 1 1, blank space, 1 1 1 0.

Back to Code

The hexadecimal number is grouped in sets of 4 hex digits. The digits are as follows: C A F E, blank space, B A B E.

Back to Code

The table has 3 columns labelled Unicode Character, Decimal Value, and Hex Value. The row entries are as follows.
Row 1. Unicode Character: N U L, the null character, left parentheses, a nonprintable character, right parentheses. Decimal Value: 0. Hex Value: 0 0 0 0.
Row 2. Unicode Character: asterisk. Decimal Value: 42. Hex Value: 0 0 2 A.

Row 3. Unicode Character: 1. Decimal Value: 49. Hex Value: 0 0 3 1.

Row 4. Unicode Character: 2. Decimal Value: 50. Hex Value: 0 0 3 2.

Row 5. Unicode Character: uppercase a. Decimal Value: 65. Hex Value: 0 0 4 1.

Row 6. Unicode Character: uppercase b. Decimal Value: 66. Hex Value: 0 0 4 2.

Row 7. Unicode Character: lowercase a. Decimal Value: 97. Hex Value: 0 0 6 1.

Row 8. Unicode Character: lowercase b. Decimal Value: 98. Hex Value: 0 0 6 2.

Row 9. Unicode Character: right brace. Decimal Value: 125. Hex Value: 0 0 7 D.

Row 10. Unicode Character: delete, left parentheses, a nonprintable character, right parentheses. Decimal Value: 127. Hex Value: 0 0 7 F.

Back to Table

Line 1: read first number. Line 2: read second number. Line 3: set total to, left parentheses, first number plus second number, right parentheses. Line 4: output total.

Back to Example

Line 1: read a number. Line 2: call the square root method, passing the number and receiving the square root. Line 3: output the square root of the number.

Back to Example

Line 1: read a number. Line 2: if the number is greater than or equal to 0. Line 3, indented once: write, double quotes, Number is positive, period, double quotes. Line 4: else. Line 5, indented once: write, double quotes, Number is negative, period, double quotes.

Back to Example

Line 1: set total to 0. Line 2: read a number. Line 3: while there was a number to read, repeat next two instructions. Line 4, indented once: add number to total. Line 5, indented once: read the next number. Line 6: write total.

Back to Example

java c Class Name, dot, java.

Back to Code

java Class Name.

Back to Code

java Class Name dot java.

Back to Code

java c, hyphen, version.

Back to Code

java c 17 point 0 point 1.

Back to Code

java, hyphen, version.

Back to Code

Line 1: java version, double quotes, 17 point 0 point 1, double quotes, 2021, hyphen, 10, hyphen, 19, L T S. Line 2: Java, left parentheses, T M, right parentheses, S E, Runtime Environment, left parentheses, build 17 point 0 point 1, plus, 12, hyphen, L T S, hyphen, 39, right parentheses. Line 3: Java Hot Spot, left parentheses, T M, right parentheses, 64, hyphen, Bit Server, V M, left parentheses, build 17 point 0 point 1, plus, 12, hyphen, L T S, hyphen, 39, comma, mixed mode, comma, sharing, right parentheses.

Back to Code

Program code. The words in the class name are merged. The code contains the keywords: public, class, static, void, main, String. Line 1: 2 forward slashes, First program in Java. Line 2: 2 forward slashes, Anderson, comma, Franceschi. Line 3: blank. Line 4: public class First Program. Line 5: left brace. Line 6, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 7, indented once: left brace. Line 8, indented twice: System dot out dot print l n, left parentheses, double quotes, Programming is not a spectator sport, exclamation mark, double quotes, right parentheses, semicolon. Line 9: blank. Line 10, indented twice: System dot exit, left parentheses, 0, right parentheses, semicolon. Line 11, indented once: right brace. Line 12: right brace.

Back to Example

Line 1: First Program dot java, colon, 8, colon, error, colon, cannot find symbol.
Line 2, indented twice: System dot out dot Print l n, left parentheses, double quotes, Programming is not a spectator sport, exclamation mark, double quotes, right parentheses, semicolon.
Line 3: a caret sign under the uppercase P of Print l n in Line 2.
Line 4, indented once: symbol, colon, method Print l n, left parentheses, String, right parentheses.
Line 5, indented once: location, colon, variable out of type Print Stream.
Line 6: 1 error.

Back to Code

Line 1: First Program dot java, colon, 6, colon, error, colon, single quote, semicolon, single quote, expected.
Line 2, indented twice: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses.
Line 3: a caret sign below the right parentheses of Line 2.
Line 4: blank.

Line 5: First Program dot java, colon, 10, colon, error, colon, left angled bracket, identifier, right angled bracket, expected.
Line 6, indented twice: System dot exit, left parentheses, 0, right parentheses, semicolon.
Line 7: a caret sign below the left parentheses of Line 6.
Line 8: blank.
Line 9: First Program dot java, colon, 10, colon, error, colon, illegal start of type.
Line 10, indented twice: System dot exit, left parentheses, 0, right parentheses, semicolon.
Line 11: a caret sign below 0 in Line 10.
Line 12: blank.
Line 13: First Program dot java, colon, 12, colon, error, colon, class, comma, interface or enum expected.
Line 14: right brace.
Line 15: a caret sign under the right brace of Line 14.
Line 16: 4 errors.

Back to Code

Line 1: Error, colon, Main method not found in class First Program, comma, please define the main method as, colon.
Line 2, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses.
Line 3: or a Java F X application class must extend java f x dot application dot Application.

Back to Code

The table has 2 columns labelled Type of Error, and Usual Causes. The row entries are as follows.
Row 1. Type of Error: Compiler errors. Usual Causes: Incorrect language syntax or misspellings.
Row 2. Type of Error: Run time errors. Usual Causes: Incorrect use of classes.
Row 3. Type of Error: Logic errors. Usual Causes: Incorrect program design or incorrect implementation of the design.

Back to Table

Dot back slash jar c f, single quote, C, colon, back slash, Users, back slash, h e r v e, back slash, Documents, back slash, My Jar dot jar, single quote, First Program dot class.

Back to Code

Dot back slash jar, hyphen, hyphen, create, blank space, hyphen, hyphen, file, single quote, C, colon, back slash, Users, back slash, h e r v e, back slash, Documents, back slash, My Jar dot jar, single quote, dot, back slash, First Program dot class.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String. Line 1: Forward slash, asterisk, An application shell. Line 2, indented once: Anderson, Franceschi. Line 3: asterisk, forward slash. Line 4: public class Shell Application. Line 5: left brace. Line 6, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. 2 forward slashes, required. Line 7, indented once: left brace. Line 8, indented twice: 2 forward slashes, write your code here. Line 9, indented once: right brace. Line 10: right brace.

Back to Example

The title of the table is Java Identifiers. The table entries are as follows. Must start with a Java letter, left parentheses, uppercase a, hyphen, uppercase z, lowercase a, hyphen, lowercase z, underscore, dollar sign, or many Unicode characters, right parentheses. Can contain an almost unlimited number of letters and, or, digits, left parentheses, 0 hyphen 9, right parentheses. Cannot contain spaces. Are case sensitive. Cannot be a Java reserved word.

Back to Table

Program code. The code contains the following keywords: public, class, static, void, String, final, double. Line 1: forward slash, asterisk, Calculate the area of a circle. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Area Of Circle. Line 6: left brace. Line 7, indented once:

public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: 2 forward slashes, define the data we know. Line 10, indented twice: final double PI, equals, 3.14159, semicolon. Line 11: blank. Line 12, indented twice: 2 forward slashes, define other data we will use. Line 13, indented twice: double radius, equals, 3.5, semicolon. Line 14: blank. Line 15, indented twice: 2 forward slashes, perform the calculation and store the result. Line 16, indented twice: double area, equals, PI, asterisk, radius, asterisk, radius, semicolon. Line 17: blank. Line 18, indented twice: 2 forward slashes. Output the result. Line 19, indented twice: System dot out dot print l n, left parentheses, double quotes, The area of the circle is, double quotes, plus, area, right parentheses, semicolon. Line 20, indented once: right brace. Line 21: right brace.

Back to Example

The syntax is as follows. data Type identifier, semicolon. 2 forward slashes. This declares one variable.

Back to Code

The syntax is as follows. Line 1: data Type identifier 1, comma, identifier 2, comma, dot, dot, dot, semicolon. 2 forward slashes. This declares multiple. Line 2, indented multiple times: 2 forward slashes. Variables of the same.

Line 3, indented multiple times: 2 forward slashes. Data type.a

Back to Code

The table has 4 columns labelled Integer data type, Size in Bytes, Minimum value, and Maximum value. The row entries are as follows.
Row 1. Integer data type: byte. Size in Bytes: 1. Minimum Value: negative 128. Maximum value: 127.
Row 2. Integer data type: short. Size in Bytes: 2. Minimum Value: negative 32,768. Maximum value: 32,767.
Row 3. Integer data type: int. Size in Bytes: 4. Minimum Value: negative 2,147,483,648. Maximum value: 2,147,483,647.
Row 4. Integer data type: long. Size in Bytes: 8. Minimum Value: negative 9,223,372,036,854,775,808. Maximum value: 9,223,372,036,854,775,807.

Back to Table

The declarations are as follows.
Line 1: int test Grade, semicolon.
Line 2: int num Players, comma, high Score, comma, dice Roll, semicolon.
Line 3: short x Coordinate, comma, y Coordinate, semicolon.
Line 4: long city Population, semicolon.
Line 5: byte age In Years, semicolon.

Back to Code

The table has 4 columns labelled Floating point data type, Size in bytes, Minimum Positive Nonzero value, and Maximum value. The row entries are as follows.
Row 1. Floating Point Data Type: float. Size in Bytes: 4. Minimum Positive Nonzero Value: 1.4 E negative 45. Maximum Value: 3.4028235 E 38.
Row 2. Floating Point Data Type: double. Size in Bytes: 8. Minimum Positive Nonzero Value: 4.9 E negative 324. Maximum Value: 1.7976931348623157 E 308.

Back to Table

The declarations are as follows.
Line 1: float sales Tax, semicolon.
Line 2: double interest Rate, semicolon.
Line 3: double paycheck, comma, sum Salaries, semicolon.

Back to Code

The table has 4 columns labelled Character Data Type, Size in Bytes, Minimum Value, and Maximum value. The row entries are as follows.
Row 1. Character data type: char. Size in Bytes: 2. Minimum Value: The character encoded as 0 0 0 0, the null character. Maximum Value: The value F F F F, which is a special code for, double quotes, not a character, double quotes.

Back to Table

The declarations are as follows.
Line 1: char final Grade, semicolon.
Line 2: char middle Initial, semicolon.
Line 3: char new line, comma, tab, comma, double Quotes, semicolon.

Back to Code

The table has 2 columns labelled Boolean Data Type and Possible Values. The row entries are as follows.
Row 1. Boolean Data Type: Boolean. Possible Values: true.
Row 2. Boolean Data Type: blank. Possible Values: false.

Back to Table

The declarations are as follows.
Line 1: boolean is Empty, semicolon.
Line 2: Boolean passed, comma, failed, semicolon.

Back to Code

The syntax is as follows. data Type variable Name equals initial Value, semicolon.

Back to Code

The syntax is as follows. data Type variable 1 equals initial Value 1, comma, variable 2 equals initial Value 2, semicolon.

Back to Code

The statement is as follows. Line 1: int test Grade equals 100, semicolon.

Back to Code

The table has 2 columns labelled Data Type and Literal Format. The row entries are as follows.
Row 1. Data Type: int, short, byte. Literal Format: Optional initial sign, left parentheses, plus or minus, right parentheses, followed by digits 0 hyphen 9 in any combination. A literal in this format is an int literal, semicolon, however, an int literal may be assigned to a byte or short variable if the literal is a legal value for the assigned data type. An integer literal that begins with a 0 digit is considered to be an octal number, left parentheses, base 8, right parentheses, and the remaining digits must be 0 hyphen 7. An integer literal that begins with 0 x is considered to be an hexadecimal number, left parentheses, base 16, right parentheses, and the remaining digits must be 0 hyphen 9 or A hyphen F.
Row 2. Data Type: long. Literal Format: Optional initial sign, left parentheses, plus or minus, right parentheses, followed by digits 0 hyphen 9 in any combination, terminated with an uppercase l or lowercase l. It’s preferable to use the capital l, because the lowercase l can be confused with the number 1. An integer literal that begins with a 0 digit is considered to be an octal number, left parentheses, base 8, right parentheses, and the remaining digits must be 0 hyphen 7. An integer literal that begins with 0 x is considered to be a hexadecimal

number, left parentheses, base 16, right parentheses, and the remaining digits must be 0 hyphen 9 or A hyphen F. Row 3. Data Type: float. Literal Format: Optional initial sign, left parentheses, plus or minus, right parentheses, followed by a floating point number in fixed or scientific format, terminated by an upper case f or lowercase f. Row 4. Data Type: double. Literal Format: Optional initial sign, left parentheses, plus or minus, right parentheses, followed by a floating point number in fixed or scientific format. Row 5. Data Type: char. Literal Format: Any printable character enclosed in single quotes. A decimal value from 0 to 65,535. Single quote, back slash, u n n n n, single quote, where n n n n are hexadecimal digits. Single quote, back slash m, single quote, where back slash m is an escape sequence. For example, single quote, back slash n, single quote, represents a newline and single quote, back slash, t, single quote, represents a tab character. Row 6. Data Type: boolean. Literal Format: true or false.

Back to Table

Line 1: j shell, right angled bracket, int test Grade, equals, 100, semicolon. Line 2: test Grade, equals, equals, right angled bracket, 100. Line 3: j shell, right angled bracket, long city Population, equals, 425612340, uppercase l, semicolon. Line 4: city Population, equals, equals, right angled bracket, 425612340. Line 5: j shell, right angled bracket, byte number Of Pets, equals, 2, semicolon. Line 6: number Of Pets, equals, equals, right angled bracket,

2. Line 7: j shell, right angled bracket, short age In Years, equals, 19, semicolon. Line 8: age In Years, equals, equals, right angled bracket, 19. Line 9: j shell, right angled bracket, float sales Tax, equals, decimal point, 0 5, uppercase f, semicolon. Line 10: sales Tax, equals, equals, right angled bracket, 0.05. Line 11: j shell, right angled bracket, double interest Rate, equals, 0.0425, semicolon. Line 12: interest Rate, equals, equals, right angled bracket, 0.0425. Line 13: j shell, right angled bracket, double avogadro Number, equals, plus, 6.022, uppercase e, 23, semicolon. Line 14: avogadro Number, equals, equals, right angled bracket, 6.022, uppercase e, 23. Line 15: j shell, right angled bracket, char final Grade, equals, single quote, uppercase a, single quote, semicolon. Line 16: final Grade, equals, equals, right angled bracket, single quote, uppercase a, single quote. Line 17: j shell, right angled bracket, boolean is Empty, equals, true, semicolon. Line 18: is Empty, equals, equals, right angled bracket, true.

Back to Example

The syntax is as follows. data Type variable 2 equals variable 1, semicolon.

Back to Code

Line 1: j shell, right angled bracket, forward slash, v a r s. Line 2: pipe, blank spaces, int test Grade equals 100. Line 3: pipe, blank spaces, long city Population equals

425612340.
Line 4: pipe, blank spaces, byte number Of Pets equals 2.
Line 5: pipe, blank spaces, short age In Years equals 19.
Line 6: pipe, blank spaces, float sales Tax equals 0.05.
Line 7: pipe, blank spaces, double interest Rate equals 0.0425.
Line 8: pipe, blank spaces, double Avogadro Number equals 6.022 E 23.
Line 9: pipe, blank spaces, char final Grade equals, single quote, uppercase a, single quote.
Line 10: pipe, blank spaces, boolean Is Empty equals true.
Line 11: j shell, right angled bracket.

Back to Code

Line 1: j shell, right angled bracket, boolean Is Passing Grade equals true, semicolon.
Line 2: is Passing Grade, 2 equals sign, right angled bracket, true.
Line 3: blank.
Line 4: j shell, right angled bracket, boolean Is Promoted equals is Passing Grade, semicolon.
Line 5: is Promoted, 2 equals sign, right angled bracket, true.

Back to Code

Line 1: j shell, right angled bracket, float sales Tax equals .05 f, semicolon.

Line 2: sales Tax, 2 equals sign, right angled bracket, 0.05.
Line 3: blank.
Line 4: j shell, right angled bracket, double tax Rate equals sales Tax, semicolon.
Line 5: tax Rate, 2 equals sign, right angled bracket, 0.05000000074505806.

Back to Code

The table has 2 columns labelled Data Type and Compatible Data Types. The row entries are as follows.
Row 1. Data Type: byte. Compatible Data Types: byte.
Row 2. Data Type: short. Compatible Data Types: byte, short.
Row 3. Data Type: int. Compatible Data Types: byte, short, int, char.
Row 4. Data Type: long. Compatible Data Types: byte, short, int, char, long.
Row 5. Data Type: float. Compatible Data Types: byte, short, int, char, long, float.
Row 6. Data Type: double. Compatible Data Types: byte, short, int, char, long, float, double.
Row 7. Data Type: boolean. Compatible Data Types: boolean.
Row 8. Data Type: char. Compatible Data Types: char.

Back to Table

The statements are as follows.

Line 1: double two Cents, semicolon.
Line 2: double two Cents equals 2, semicolon. 2 forward slashes. Incorrect, second declaration of two Cents.

Back to Code

The statements are as follows.
Line 1: double cash In Hand, semicolon.
Line 2: int cash In Hand, semicolon. 2 forward slashes. Incorrect, data type cannot be changed.

Back to Code

Line 1: double quotes, Hello, double quotes.
Line 2: double quotes, Hello, blank space, world, double quotes.
Line 3: double quotes, The value of x is, blank space, double quotes.

Back to Code

The statement is as follows. Line 1: System dot out dot print l n, left parentheses, double quotes, Never pass a water fountain. Line 2, indented multiple times: without taking a drink, dot, double quotes, right parentheses, semicolon.

Back to Code

Line 1: j shell, right angled bracket, blank space, System dot out dot print l n, left parentheses, double quotes, Never pass a water fountain.

Line 2: pipe, blank space, Error, colon.
Line 3: pipe, blank space, unclosed string literal.
Line 4: pipe, blank space, System dot out dot print l n, Never pass a water fountain.
Line 5: pipe followed by multiple indents. A caret symbol is printed below the double quotes in Line 4.

Back to Code

Line 1: j shell, right angled bracket, blank space, System dot out dot print l n, left parentheses, double quotes, Never pass a water fountain, double quotes.
Line 2: dot, dot, dot, right angled bracket. Multiple indents. Plus, double quotes, without taking a drink, dot, double quotes, right parentheses, semicolon.
Line 3: Never pass a water fountain without taking a drink, dot.

Back to Code

Line 1: System dot out dot print l n, left parentheses, double quotes, She said, comma, double quotes, Java is fun, double quotes, dot, double quotes, right parentheses, semicolon. 2 forward slashes. Illegal quotes.
Line 2, indented multiple times: 2 forward slashes. Within literal.

Back to Code

Line 1: j shell, right angled bracket, blank space, System dot out dot print l n, left parentheses, double quotes, She

said, comma, double quotes, Java is fun, double quotes, dot, double quotes, right parentheses, semicolon.
Line 2: pipe, blank space, Error, colon.
Line 3: pipe, single quote, right parentheses, single quote, expected.
Line 4: pipe, System dot out dot print l n, left parentheses, double quotes, She said, comma, double quotes, Java is fun, double quotes, dot, double quotes, right parentheses, semicolon.
Line 5: pipe, multiple indents. Caret sign under the uppercase j character in line 4.

Back to Code

The table has 2 columns labelled Character and Escape Sequence. The row entries are as follows.
Row 1. Character: newline. Escape Sequence: backslash n.
Row 2. Character: tab. Escape Sequence: backslash t.
Row 3. Character: double quotes. Escape Sequence: backslash, double quotes.
Row 4. Character: single quote. Escape Sequence: backslash, single quote.
Row 5. Character: backslash. Escape Sequence: 2 backslashes.
Row 6. Character: backspace. Escape Sequence: backslash b.
Row 7. Character: carriage return. Escape Sequence: backslash r.
Row 8. Character: form feed. Escape Sequence:

backslash f.

Back to Table

Line 1: j shell, right angled bracket, blank space, System dot out dot print l n, left parentheses, double quotes, She said, comma, back slash, double quotes, Java is fun, back slash, double quotes, dot, double quotes, right parentheses, semicolon.
Line 2: She said, comma, double quotes, Java is fun, double quotes, dot.

Back to Code

Line 1: j shell, right angled bracket, System dot out dot print l n, left parentheses, double quotes, One potato, back slash, n, Two potatoes, double quotes, right parentheses, semicolon.
Line 2: One potato Two potatoes.

Back to Code

Line 1: j shell, right angled bracket, System dot out dot print l n, left parentheses, double quotes, backslash, t, Tabs can make the output easier to read, dot, double quotes, right parentheses, semicolon.
Line 2, indented once: Tabs can make the output easier to read, dot.

Back to Code

Program code. The code contains the following keywords:

public, class, static, void, String. Line 1: forward slash, asterisk, Using Text Blocks. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Text Block. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: String menu, equals, 3 double quotes. Line 10, indented thrice: Please choose a language, colon. Line 11, indented thrice: back slash, t 1, hyphen, Java. Line 12, indented thrice: back slash, t 2, hyphen, Python. Line 13, indented thrice: back slash, t 3, hyphen, C, plus, plus. Line 14, indented thrice: 3 double quotes, semicolon. Line 15: blank. Line 16, indented twice: System dot out dot print l n, left parentheses, menu, right parentheses, semicolon. Line 17, indented once: right brace. Line 18: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, final, char, double, int. Line 1: forward slash, asterisk, Constants Class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Constants. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: final char ZORRO, equals, single quote, uppercase z, single quote, semicolon. Line 10, indented twice: final double PI, equals, 3.14159, semicolon. Line 11, indented twice: final int DAYS underscore IN underscore LEAP underscore YEAR, equals, 366, comma, DAYS underscore IN underscore NON underscore LEAP underscore YEAR, equals, 365, semicolon. Line 12: blank. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, The value of constant ZORRO is, double quotes, plus, ZORRO, right parentheses, semicolon. Line 14, indented twice: System dot out dot print l n, left parentheses, double quotes, The value of constant PI is, double quotes, plus, PI, right parentheses, semicolon. Line 15, indented twice: System dot out dot print l n, left parentheses, double quotes, The number of days in a leap year is, double quotes. Line 16, indented multiple times: plus, DAYS underscore IN underscore LEAP underscore YEAR, right parentheses, semicolon. Line 17, indented twice: System dot out dot print l n, left parentheses, double quotes, The number of days in a non, hyphen, leap year is, double quotes. Line 18, indented multiple times: plus, DAYS underscore IN underscore NON underscore LEAP underscore YEAR, right parentheses, semicolon. Line 19: blank. Line 20, indented twice: 2 forward slashes, PI, equals, 3.14, semicolon. Line 21, indented twice: 2 forward slashes, The statement above would generate a compiler error. Line 22, indented twice: 2 forward slashes, You cannot change the value of a constant. Line 23, indented once: right brace. Line 24: right brace.

Back to Example

The syntax is as follows. target equals expression, semicolon.

Back to Code

Line 1: j shell, right angled bracket, int number Of Players equals 10, semicolon.
Line 2: number Of Players, 2 equals sign, right angled bracket, 10.
Line 3: j shell, right angled bracket, number Of Players equals 8, semicolon.
Line 4: number Of Players, 2 equals sign, right angled bracket, 8.
Line 5: j shell, right angled bracket, forward slash, vars.
Line 6: pipe, single indent, int number Of Players, equals, 8.

Back to Code

Line 1: j shell, right angled bracket, int legal Age equals 18, semicolon.
Line 2: legal Age, 2 equals sign, right angled bracket, 18.
Line 3: j shell, right angled bracket, int voter Age equals legal Age, semicolon.
Line 4: voter Age, 2 equals sign, right angled bracket, 18.

Back to Code

The declarations are as follows.

Line 1: int length equals width asterisk 2, semicolon. 2 forward slashes. Invalid, comma, width is not yet defined.
Line 2: int width equals 30, semicolon.

Back to Code

Line 1: j shell, right angled bracket, int length equals width asterisk 2, semicolon.
Line 2: pipe, blank space, Error, colon.
Line 3: pipe, blank space, cannot find symbol.
Line 4: pipe, tab space, symbol, colon, variable width.
Line 5: pipe, blank space, int length equals width asterisk 2, semicolon.
Line 6: pipe, multiple indents, caret sign below lowercase w in line 5.

Back to Code

Row 1. Operator: plus sign. Operation: addition.
Row 2. Operator: minus sign. Operation: subtraction.
Row 3. Operator: asterisk. Operation: multiplication.
Row 4. Operator: forward slash. Operation: division.
Row 5. Operator: percentage sign. Operation: modulus, left square bracket, remainder after division, right square bracket.

Back to Table

Program code. The code contains the following keywords: public, class, static, void, double, String, int. Line 1: forward slash, asterisk, Arithmetic Operators. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Arithmetic Operators. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: 2 forward slashes, calculate the cost of lunch. Line 10, indented twice: double salad, equals, 5.95, semicolon. Line 11, indented twice: double water, equals, decimal point, 89, semicolon. Line 12, indented twice: System dot out dot print l n, left parentheses, double quotes, The cost of lunch is, dollar sign, double quotes. Line 13, indented multiple times: plus, left parentheses, salad, plus, water, right parentheses, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: 2 forward slashes, calculate your age as of a certain year. Line 16, indented twice: int target Year, equals, 2025, semicolon. Line 17, indented twice: int birth Year, equals, 2005, semicolon. Line 18, indented twice: System dot out dot print l n, left parentheses, double quotes, Your age in, double quotes, plus, target Year, plus, double quotes, is, double quotes. Line 19, indented multiple times: plus, left parentheses, target Year, minus, birth Year, right parentheses, right parentheses, semicolon. Line 20: blank. Line 21, indented twice: 2 forward slashes, calculate the total calories of apples. Line 22, indented twice: int calories Per Apple, equals, 127, semicolon. Line 23, indented twice: int number Of Apples, equals, 3, semicolon. Line 24, indented twice: System dot out dot print l n, left parentheses, double quotes, The calories in, double quotes, plus, number Of Apples. Line 25, indented multiple times: plus, double quotes, apples is, double quotes, plus. Line 26, indented multiple times: plus, left parentheses, calories Per Apple, asterisk, number Of Apples, right parentheses, right parentheses, semicolon. Line 27: blank. Line 28, indented twice: 2 forward slashes, calculate miles per gallon. Line 29, indented twice: double miles, equals, 426.8, semicolon. Line 30, indented twice: double gallons, equals, 15.2, semicolon. Line 31, indented twice: double mileage, equals, miles, forward slash, gallons, semicolon. Line 32, indented twice: System dot out dot print l n, left parentheses, double quotes, The mileage is, double quotes. Line 33, indented multiple times: plus, mileage, plus, double quotes, miles per gallon, period, double quotes, right parentheses, semicolon. Line 34, indented once: right brace. Line 35: right brace.

Back to Example

Line 1: equals, 2 asterisk, 25, plus, 3, asterisk, 10, plus, 2, asterisk, 5.
Line 2: equals, blank space, 50, blank space, plus, 3, asterisk, 10, plus, 2, asterisk, 5.
Line 3: equals, blank spaces, 53, blank spaces, asterisk, 10, plus, 2, asterisk, 5.
Line 4: equals, blank spaces, 530, blank spaces, plus, 2, asterisk, 5.
Line 5: equals, blank spaces, 532, asterisk, 5.

Line 6: equals, blank spaces, 2660.

Back to Code

The table has 3 columns labelled Operator Hierarchy, Order of Same Statement Evaluation, and Operation. The row entries are as follows.
Row 1. Operator Hierarchy: left parentheses, right parentheses. Order of Same Statement Evaluation: left to right. Operation: parentheses for explicit grouping.
Row 2. Operator Hierarchy: asterisk, forward slash, percentage sign. Order of Same Statement Evaluation: left to right. Operation: multiplication, division, modulus.
Row 3. Operator Hierarchy: plus, minus. Order of Same Statement Evaluation: left to right. Operation: addition, subtraction.
Row 4. Operator Hierarchy: equals sign. Order of Same Statement Evaluation: right to left. Operation: assignment.

Back to Table

Line 1: int pennies equals 2 asterisk 25 plus 3 asterisk 10 plus 2 asterisk 5, semicolon.
Line 2, indented twice: equals, 50 plus 30 plus 10.
Line 3, indented twice: equals, 90.

Back to Code

Line 1: int pennies equals, left parentheses, 2 asterisk 25, right parentheses, plus, left parentheses, 3 asterisk 10, right parentheses, plus, left parentheses, 2 asterisk 5,

right parentheses, semicolon.
Line 2, indented once: equals, 50, plus, 30, plus, 10.
Line 3, indented once: equals 90.

Back to Code

Line 1: j shell, right angled bracket, int pennies equals 2 asterisk 25 plus 3 asterisk 10 plus 2 asterisk 5, semicolon.
Line 2: pennies, 2 equals sign, right angled bracket, 90.
Line 3: j shell, right angled bracket, pennies, equals, left parentheses, 2 asterisk 25, right parentheses, plus, left parentheses, 3 asterisk 10, right parentheses, plus, left parentheses, 2 asterisk 5, right parentheses, semicolon.
Line 4: pennies, 2 equals sign, right angled bracket, 90.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String. Line 1: Forward slash, asterisk. Metric Length, hyphen, converts inches to centimeters. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Metric Length. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9: blank. Line 10, indented twice: Forward slash, 5 asterisks, 1, period, What data values do we know, question mark. Line 11, indented thrice: We know that there are 2.54 centimeters in an inch. Line 12, indented thrice: Declare a double constant named C M underscore PER underscore INCH. Line 13, indented thrice: Assign C M underscore PER underscore INCH the value 2.54. Line 14, indented twice: asterisk, forward slash. Line 15: blank. Line 16: blank. Line 17, indented twice: forward slash, 5 asterisks, 2. What other data does the program require, question mark. Line 18, indented thrice: For this program, comma, we require the number of inches. Line 19, indented thrice: Declare a double variable named inches. Line 20, indented thrice: Assign any desired value to this variable. Line 21, indented twice: asterisk, forward slash. Line 22: blank. Line 23: blank. Line 24, indented twice: Forward slash, 5 asterisks, 3. Calculation, colon, convert inches to centimeters. Line 25, indented thrice: Declare a double variable named centimeters. Line 26, indented thrice: Multiply inches by C M underscore PER underscore INCH. Line 27, indented thrice: and store the result in centimeters. Line 28, indented twice: asterisk, forward slash. Line 29: blank. Line 30: blank. Line 31, indented twice: Forward slash, 5 asterisks. 4. Output. Line 32, indented thrice: Write one or two statements that output. Line 33, indented thrice: the original inches and the equivalent centimeters. Line 34, indented thrice: Try to match the sample output in Figure 2.5. Line 35, indented twice: asterisk, forward slash. Line 36: blank. Line 37: blank. Line 38: blank. Line 39, indented once: right brace. Line 40: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, final, int, double. Line 1: Forward slash, asterisk. Division And Modulus Class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: public class Division And Modulus. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: final int PENNIES underscore PER underscore QUARTER, equals 25, semicolon. Line 10, indented twice: int pennies, equals, 113, semicolon. Line 11: blank. Line 12, indented twice: int quarters, equals, pennies, forward slash, PENNIES underscore PER underscore QUARTER, semicolon. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, There are, double quotes, plus, quarters, plus, double quotes, blank space, quarters in, blank space, double quotes. Line 14, indented multiple times: plus, pennies, plus, double quotes, blank space, pennies, double quotes, right parentheses, semicolon. Line 15: blank. Line 16, indented twice: int pennies Left Over, equals, pennies, percentage sign, PENNIES underscore PER underscore QUARTER, semicolon. Line 17, indented twice: System dot out dot print l n, left parentheses, double quotes, There are, blank space, double quotes, plus, pennies Left Over. Line 18, indented multiple times: plus, double quotes, blank space, pennies left over, double quotes, right parentheses, semicolon.

Line 19: blank. Line 20, indented twice: final double MONTHS underscore PER underscore YEAR, equals, 12, semicolon. Line 21, indented twice: double annual Salary, equals, 50000.0, semicolon. Line 22: blank. Line 23, indented twice: double monthly Salary, equals, annual Salary, forward slash, MONTHS underscore PER underscore YEAR, semicolon. Line 24, indented twice: System dot out dot print l n, left parentheses, double quotes, The monthly salary is, double quotes, plus, monthly Salary, right parentheses, semicolon. Line 25, indented once: right brace. Line 26: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, double, int. Line 1: Forward slash, asterisk. Division by Zero Class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: public class Division By Zero. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: double result, equals, 4.3, forward slash, 0.0, semicolon. Line 10, indented twice: System dot out dot print l n, left parentheses, double quotes, The value of result 1 is, blank space, double quotes, plus, result 1, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: double result 2, equals, 0.0, forward slash, 0.0, semicolon. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, The value of result 2 is, double quotes, plus, result 2, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: int result 3, equals, 4, forward slash, 0, semicolon. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, The value of result 3 is, double quotes, plus, result 3, right parentheses, semicolon. Line 17, indented once: right brace. Line 18: right brace.

Back to Example

The table has 4 columns labelled Data Type of One Operand, Data Type of Other Operand, Promotion of Other Operand, and Data Type of Result. The row entries are as follows.
Row 1. Data Type of One Operand: double. Data Type of Other Operand: char, byte, short, int, long, float. Promotion of Other Operand: double. Data Type of Result: double.
Row 2. Data Type of One Operand: float. Data Type of Other Operand: char, byte, short, int, long. Promotion of Other Operand: float. Data Type of Result: float.
Row 3. Data Type of One Operand: long. Data Type of Other Operand: char, byte, short, int. Promotion of Other Operand: long. Data Type of Result: long.
Row 4. Data Type of One Operand: int. Data Type of Other Operand: char, byte, short. Promotion of Other Operand: int. Data Type of Result: int.
Row 5. Data Type of One Operand: short. Data Type of

Other Operand: char, byte. Promotion of Other Operand: Both operands are promoted to int. Data Type of Result: int.
Row 6. Data Type of One Operand: byte. Data Type of Other Operand: char. Promotion of Other Operand: Both operands are promoted to int. Data Type of Result: int.

Back to Table

Line 1: j shell, right angled bracket, int radius equals 4, semicolon.
Line 2: radius, 2 equals sign, right angled bracket, 4.
Line 3: j shell, right angled bracket, int radius Squared equals radius asterisk radius, semicolon.
Line 4: radius Squared, 2 equals sign, right angled bracket, 16.
Line 5: j shell, right angled bracket, double area equals 3.14159 asterisk radius Squared, semicolon.
Line 6: area, 2 equals sign, right angled bracket, 50.26544.
Line 7: j shell, right angled bracket, forward slash, vars.
Line 8: pipe, blank spaces, int radius equals 4.
Line 9: pipe, blank spaces, int radius Squared equals 16.
Line 10: pipe, blank spaces, double area equals 50.26544.

Back to Code

Line 1: j shell, right angled bracket, int total Scores equals 94 plus 86 plus 88 plus 97, semicolon.

Line 2: total Scores, 2 equals sign, right angled bracket, 365.
Line 3: blank.
Line 4: j shell, right angled bracket, int count equals 4, semicolon.
Line 5: count, 2 equals sign, right angled bracket, 4.
Line 6: blank.
Line 7: j shell, right angled bracket, total Scores, forward slash, count.
Line 8: dollar sign, 3, 2 equals sign, right angled bracket, 91.
Line 9: blank.
Line 10: j shell, right angled bracket, double average equals total Scores forward slash count, semicolon.
Line 11: average, 2 equals sign, right angled bracket, 91.0.

Back to Code

Line 1: j shell, right angled bracket, left parentheses, double, right parentheses, left parentheses, total Scores, forward slash, count, right parentheses.
Line 2: dollar sign, 7, 2 equals sign, right angled bracket, 91.0.
Line 3: blank.
Line 4: j shell, right angled bracket, average, equals, left parentheses, double, right parentheses, left parentheses, total Scores, forward slash, count, right parentheses, semicolon.
Line 5: average, 2 equals sign, right angled bracket, 91.0.

Back to Code

Line 1: j shell, right angled bracket, left parentheses, double, right parentheses, total Scores, forward slash, count.
Line 2: dollar sign, 9, 2 equals sign, right angled bracket, 91.25.
Line 3: blank.
Line 4: j shell, right angled bracket, average, equals, left parentheses, double, right parentheses, total Scores, forward slash, count, semicolon.
Line 5: average, 2 equals sign, right angled bracket, 91.25.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, int. Line 1: Forward slash, asterisk. Shortcut Operators Class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: public class Shortcut Operators. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: int a, equals, 6, semicolon. Line 10, indented once: int b, equals, 2, semicolon. Line 11: blank. Line 12, indented once: System dot out dot print l n, left parentheses, double quotes, At the begining, comma, a is, blank space, double quotes, plus, a, right parentheses, semicolon. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, Increment a with prefix notation, colon, blank space, double quotes, plus, 2 plus symbols, a, right parentheses, semicolon. Line 14, indented twice: System dot out dot print l n, left parentheses, double quotes, In the end, comma, a is, blank space, double quotes, plus, a, right parentheses, semicolon. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, At the begining, comma, b is, blank space, double quotes, plus, b, right parentheses, semicolon. Line 17, indented twice: System dot out dot print l n, left parentheses, double quotes, Increment b with postfix notation, colon, blank space, double quotes, plus, b, plus, plus, right parentheses, semicolon. Line 18, indented twice: System dot out dot print l n, left parentheses, double quotes, In the end, comma, b is, blank space, double quotes, plus, b, right paentheses, semicolon. Line 19, indented once: right brace. Line 20: right brace.

Back to Example

The statements are as follows.
Line 1: int length equals 3, semicolon. 2 forward slashes. Length in feet.
Line 2: length asterisk equals 12, semicolon. 2 forward slashes. Length converted to inches.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, int. Line 1: Forward slash, asterisk. Shortcut Arithmetic Operators. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Shortcut Arithmetic Operators. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: int a, equals, 5, semicolon. Line 10, indented twice: System dot out dot print l n, left parentheses, double quotes, a is, blank space, double quotes, plus, a, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: a, plus, equals, 10, semicolon. 2 forward slashes. a equals, a plus 10, semicolon. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, After a, plus, equals, 10, semicolon, a is, blank space, double quotes, plus, a, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: a, minus, equals, 3, semicolon. 2 forward slashes. a, equals, a minus 3, semicolon. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, After a, minus, equals, 3, semicolon, a is, blank space, double quotes, plus, a, right parentheses, semicolon. Line 17: blank. Line 18, indented twice: a, asterisk, equals, 2, semicolon. 2 forward slashes, a, equals, a asterisk 2, semicolon. Line 19, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash n, After a asterisk, equals, 2, semicolon, a is, blank space, double quotes, plus, a, right parentheses, semicolon. Line 20: blank. Line 21, indented twice: a, forward slash, equals, 6, semicolon. 2 forward slashes. a, equals, a, forward slash, 6, semicolon. Line 22, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash n, After a, forward slash, equals, 6, semicolon, a is, blank space, double quotes, plus, a, right parentheses, semicolon. Line 23: blank. Line 24, indented twice: a, percentage sign, equals, 3, semicolon. 2 forward slashes, a, equals, a, percentage sign, 3, semicolon. Line 25, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash n, After a, percentage sign, equals, 3, semicolon, a is, blank space, double quotes, plus, a, right parentheses, semicolon. Line 26, indented once: right brace. Line 27: right brace.

Back to Example

The table has 3 columns labelled Shortcut Operator, Example, and Equivalent Statement. The row entries are as follows.
Row 1. Shortcut Operator: plus plus. Example: a plus plus, semicolon, or, plus plus a, semicolon. Equivalent Statement: a equals a plus 1, semicolon.
Row 2. Shortcut Operator: minus minus. Example: a minus minus, semicolon, or, minus minus a, semicolon. Equivalent Statement: a equals a minus 1, semicolon.
Row 3. Shortcut Operator: plus equals. Example: a plus equals 3, semicolon. Equivalent Statement: a equals a

plus 3, semicolon.
Row 4. Shortcut Operator: minus equals. Example: a minus equals 10, semicolon. Equivalent Statement: a equals a minus 10, semicolon.
Row 5. Shortcut Operator: asterisk equals. Example: a asterisk equals 4, semicolon. Equivalent Statement: a equals a asterisk 4, semicolon.
Row 6. Shortcut Operator: forward slash equals. Example: a forward slash equals 7, semicolon. Equivalent Statement: a equals a forward slash 7, semicolon.
Row 7. Shortcut Operator: percentage equals. Example: a percentage equals 10, semicolon. Equivalent Statement: a equals a percentage 10, semicolon.

Back to Table

The table has 3 columns labelled Operator Hierarchy, Order of Same Statement Evaluation, and Operation. The row entries are as follows.
Row 1. Operator Hierarchy: left parentheses, right parentheses. Order of Same Statement Evaluation: left to right. Operation: parentheses for explicit grouping.
Row 2. Operator Hierarchy: plus plus, minus minus. Order of Same Statement Evaluation: right to left. Operation: shortcut post increment.
Row 3. Operator Hierarchy: plus plus minus minus. Order of Same Statement Evaluation: right to left. Operation: shortcut pre increment.
Row 4. Operator Hierarchy: asterisk, forward slash, percentage sign. Order of Same Statement Evaluation:

left to right. Operation: multiplication, division, modulus.
Row 5. Operator Hierarchy: plus, minus. Order of Same Statement Evaluation: left to right. Operation: addition or String concatenation, subtraction.
Row 6. Operator Hierarchy: equals, plus equals, minus equals, asterisk equals, forward slash equals, percentage equals. Order of Same Statement Evaluation: right to left. Operation: assignment operator and shortcut assignment operators.

Back to Table

Program code. The code contains the following keywords: public, class, static, void, String. Line 1: Forward slash, asterisk. Temperature Conversion. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Temperature Conversion. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: 2 forward slashes. 5 asterisks. 1. declare any constants here. Line 10: blank. Line 11: blank. Line 12, indented twice: 2 forward slashes. 5 asterisks. 2. declare the temperature in Fahrenheit as an int. Line 13: blank. Line 14: blank. Line 15, indented twice: 2 forward slashes. 5 asterisks. 3. Calculate equivalent Celsius temperature. Line 16: blank. Line 17: blank. Line 18, indented twice: 2 forward slashes. 5 asterisks. 4. output the temperature in Celsius. Line 19: blank. Line 20: blank.

Line 21, indented twice: 2 forward slashes. 5 asterisks. 5. convert Celsius temperature back to Fahrenheit. Line 22: blank. Line 23: blank. Line 24, indented twice: 2 forward slashes. 5 asterisks. 6. output Fahrenheit temperature to check correctness. Line 25: blank. Line 26: blank. Line 27, indented once: right brace. Line 28: right brace.

Back to Example

The syntax is as follows. data Type variable 1 equals initial Value 1, semicolon.

Back to Code

The syntax is as follows. final data Type CONSTANT underscore IDENTIFIER equals initial Value, semicolon.

Back to Code

Line 1: double a equals 12.5, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: int a equals 6, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: float a equals 13 f, semicolon.

Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: double a equals 13 forward slash 5, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: int a equals 13 forward slash 5, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: int a equals 13 percentage sign 5, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: int a equals 12 forward slash 6 asterisk 2, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: int a equals 12, forward slash, left parentheses, 6 asterisk 2, right parentheses, semicolon.

Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: int a equals 4 plus 6 forward slash 2, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: int a equals left parentheses, 4 plus 6, right parentheses, forward slash 2, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: double a equals 12.0 forward slash 5, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: int a equals left parentheses, int, right parentheses, 12.0 forward slash 5, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: double a equals left parentheses, double, right

parentheses, left parentheses, 12, right parentheses, forward slash, 5, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: double a equals left parentheses, double, right parentheses, left parentheses, 12 forward slash 5, right parentheses, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: int a equals 5, semicolon.
Line 2: a plus plus, semicolon.
Line 3: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: int a equals 5, semicolon.
Line 2: System dot out dot print l n, left parentheses, a minus minus, right parentheses, semicolon.

Back to Code

Line 1: int a equals 5, semicolon.
Line 2: System dot out dot print l n, left parentheses, minus minus a, right parentheses, semicolon.

Back to Code

Line 1: int a equals 5, semicolon.
Line 2: a plus equals 2, semicolon.
Line 3: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: int a equals 5, semicolon.
Line 2: a forward slash equals 6, semicolon.
Line 3: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: int a equals 3, semicolon. Line 2: int b equals 5, semicolon. Line 3: int c equals 8, semicolon. Line 4: 2 forward slashes. your code goes here.

Back to Code

Line 1: int a equals 3, semicolon.
Line 2: int b equals 5, semicolon.
Line 3: 2 forward slashes. your code goes here.

Back to Code

Line 1: int a equals 10, semicolon.
Line 2: int b equals 3, semicolon.
Line 3: 2 forward slashes. your code goes here.

Back to Code

Line 1: Test dot java, colon, 8, error, colon, incompatible types, colon, possible lossy conversion from double.
Line 2: to int.
Line 3, indented once: int a equals 26.4, semicolon.
Line 4: a caret sign below character 2 in line 3.
Line 5: 1 error.

Back to Code

Line 1: Test dot java, colon, 8, error, colon, single quote, semicolon, single quote, expected.
Line 2: int a equals 3.
Line 3: a caret sign below the character 3 in line 2.

Back to Code

Line 1: int a equals 32, semicolon.
Line 2: int b equals 10, semicolon.
Line 3: double c equals a forward slash b, semicolon.
Line 4: System dot out dot print l n, left parentheses, double quotes, The value of c is, double quotes, plus, c, right parentheses, semicolon.

Back to Code

Line 1: int a equals 5, semicolon.
Line 2: a equals plus 3, semicolon.
Line 3: System dot out dot print l n, left parentheses, double quotes, The value of a is, double quotes, plus a,

right parentheses, semicolon.

Back to Code

Line 1: asterisk, 3 blank spaces, asterisk.
Line 2: blank space, asterisk, blank space, asterisk, blank space.
Line 3: 2 blank spaces, asterisk, 2 blank spaces.
Line 4: blank space, asterisk, blank space, asterisk, blank space.
Line 5: asterisk, 3 blank spaces, asterisk.

Back to Code

Line 1: asterisk, 3 blank spaces, asterisk.
Line 2: blank space, asterisk, blank space, asterisk, blank space.
Line 3: 2 blank spaces, asterisk, 2 blank spaces.
Line 4: blank space, asterisk, blank space, asterisk, blank space.
Line 5: asterisk, 3 blank spaces, asterisk.

Back to Code

The text magnified by the lens reads: Look It Up.

Back to Figure

2 types of constructors are shown in the table. 1. String, left parentheses, String s t r, right parentheses. Allocates a String object with the value of s t r, which can be a String object or a String literal. 2. String, left parentheses,

right parentheses. Allocates an empty String object.

Back to Table

Program code. The code contains the following keywords: public, class, static, void, String. Line 1: Forward slash, asterisk. A Demonstration of Using Constructors. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: public class Constructors. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: String empty String, semicolon. Line 10, indented twice: empty String, equals, new String, left parentheses, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: String greeting, semicolon. Line 13, indented twice: greeting, equals, new String, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: String first Name, equals, new String, left parentheses, double quotes, Jane, double quotes, right parentheses, semicolon. Line 16, indented once: right brace. Line 17: right brace.

Back to Example

The syntax is as follows. object Reference equals new Class Name, left parentheses, argument list, right parentheses, semicolon.

Back to Code

The table has 2 columns labelled Return Value and Method name and argument list. The row entries are as follows.
Line 1. Return Value: int. Method name and argument list: length, left parentheses, right parentheses, returns the length of the String.
Line 2. Return Value: String. Method name and argument list: to Upper Case, left parentheses, right parentheses, returns a copy of the String with all letters in uppercase.
Line 3. Return Value: String. Method name and argument list: to Lower Case, left parentheses, right parentheses, returns a copy of the String with all letters in lowercase.
Line 4. Return Value: char. Method name and argument list: char At, left parentheses, int index, right parentheses, returns the character at the position specified by index.
Line 5. Return Value: int. Method name and argument list: index Of, left parentheses, String search String, right parentheses, returns the index of the beginning of the first occurrence of search String or negative 1 if search String is not found.
Line 6. Return Value: int. Method name and argument list: index Of, left parentheses, char search Char, right parentheses, index Of, left parentheses, char search Char, right parentheses, returns the index of the first occurrence of search Char in the String or negative 1 if search Char is not found.
Line 7. Return Value: String. Method name and argument

list: substring, left parentheses, int start Index, comma, int end Index, right parentheses, returns a substring of the String object beginning at the character at index start Index and ending at the character at index end Index minus 1.
Line 8. Return Value: String. Method name and argument list: substring, left parentheses, int start Index, right parentheses, returns a substring of the String object beginning at the character at index start Index and continuing to the end of the String.
Line 9. Return Value: boolean. Method name and argument list: is Blank, left parentheses, right parentheses, returns true if this String is empty or only contains white space character, left parentheses, including back slash, t, or back slash, n, right parentheses.
Line 10. Return Value: String. Method name and argument list: strip, left parentheses, right parentheses, returns a copy of this String with all the leading and trailing white characters removed, left parentheses, including back slash t, or back slash, n, right parentheses.

Back to Table

The syntax is as follows. object Reference dot method Name, left parentheses, argument list, right parentheses, or, object Reference dot method Name, left parentheses, a r g 1, comma, a r g 2, comma, a r g 2, comma, dot, dot, dot, right parentheses.

Back to Code

The rows are labelled Character and Index. The column entries are as follows.
Column 1. Character: uppercase h. Index: 0.
Column 2. Character: e. Index: 1.
Column 3. Character: l. Index: 2.
Column 4. Character: l. Index: 3.
Column 5. Character: o. Index: 4.

Back to Table

System dot out dot print l n, left parentheses, double quotes, In caps, colon, blank space, double quotes, plus, greeting dot to Upper Case, left parentheses, right parentheses, right parentheses, semicolon.

Back to Code

System dot out dot print l n, left parentheses, double quotes, In caps, colon, blank space, double quotes, plus, double quotes, Good Morning, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, int, new, null. Line 1: Forward slash, asterisk. A Demonstration of Trying to Use a Null Object Reference. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Null Reference 2. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: String greeting, equals, new String, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 10, indented twice: int len, equals, greeting dot length, left parentheses, right parentheses, semicolon. Line 11, indented twice: System dot out dot print l n, left parentheses, greeting, plus, double quotes, blank space, has, blank space, double quotes, plus, len, plus, double quotes, blank space, characters, double quotes, right parentheses, semicolon. Line 12: blank. Line 13, indented twice: greeting, equals, null, semicolon. Line 14, indented twice: 2 forward slashes. attempt to use null object reference. Line 15, indented twice: len, equals, greeting dot length, left parentheses, right parentheses, semicolon. Line 16, indented once: right brace. Line 17: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, int, new, null. Line 1: Forward slash, asterisk. A Demonstration of Trying to Use a Null Object Reference. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Null Reference 2. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: String greeting, equals, new String, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 10, indented twice: int len, equals, greeting dot length, left parentheses, right parentheses, semicolon. Line 11, indented twice: System dot out dot print l n, left parentheses, greeting, plus, double quotes, blank space, has, blank space, double quotes, plus, len, plus, double quotes, blank space, characters, double quotes, right parentheses, semicolon. Line 12: blank. Line 13, indented twice: greeting, equals, null, semicolon. Line 14, indented twice: 2 forward slashes. attempt to use null object reference. Line 15, indented twice: len, equals, greeting dot length, left parentheses, right parentheses, semicolon. Line 16, indented once: right brace. Line 17: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, new, int. Line 1: Forward slash, asterisk. Demonstrating String methods. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: public class String Demo. Line 5: left brace. Line 6, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 7, indented once: left brace. Line 8, indented twice: String s

1, equals, new String, left parentheses, double quotes, O O P in Java, blank space, double quotes, right parentheses, semicolon. Line 9, indented twice: System dot out dot print l n, left parentheses, double quotes, s 1 is, colon, blanks pace, double quotes, plus, s 1, right parentheses, semicolon. Line 10, indented twice: String s 2, equals, double quotes, is not that difficult, period, double quotes, semicolon. Line 11, indented twice: System dot out dot print l n, left parentheses, double quotes, s 2 is, colon, blank space, double quotes, plus, s 2, right parentheses, semicolon. Line 12: blank. Line 13, indented twice: String s 3, equals, s 1, plus, s 2, semicolon. 2 forward slashes, new String is s 1, comma, followed by s 2. Line 14, indented twice: System dot out dot print l n, left parentheses, double quotes, s 1, plus, s 2, is, colon, blank space, double quotes, plus, s 3, right parentheses, semicolon. Line 15: blank. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, s 1 is still, colon, blank space, double quotes, plus, s 1, right parentheses, semicolon. Line 17, indented twice: System dot out dot print l n, left parentheses, double quotes, s 2 is still, colon, blank space, double quotes, plus, s 2, right parentheses, semicolon. 2 forward slashes, s 2 is unchanged. Line 18: blank. Line 19, indented twice: String empty, equals, new String, left parentheses, right parentheses, semicolon. Line 20, indented twice: System dot out dot print l n, left parentheses, double quotes, back slash, n, The length of the empty string is, blank space, double quotes. Line 21, indented multiple times: plus, empty dot length, left parentheses, right parentheses, right parentheses, semicolon. Line 22: blank. Line 23, indented twice: String greeting, equals, double quotes, Hello, double quotes, semicolon. 2 forward slashes, instantiate greeting. Line 24, indented twice: char first Char, equals, greeting dot char At, left parentheses, 0, right parentheses, semicolon. Line 25, indented twice: char last Char, equals, greeting dot char At, left parentheses, greeting dot length, left parentheses, right parentheses, minus, 1, right parentheses, semicolon. Line 26, indented twice: System dot out dot print l n, left parentheses, double quotes, back slash, n, The first and last characters of, blank space, double quotes. Line 27, indented multiple times: plus, greeting, plus, double quotes, blank space, are, blank space, double quotes, plus, first Char. Line 28, indented multiple times: plus, double quotes, blank space, and, blank space, double quotes, plus, last Char, right parentheses, semicolon. Line 29: blank. Line 30, indented twice: int index Of E, equals, greeting dot index Of, left parentheses, single quote, e, single quote, right parentheses, semicolon. Line 31, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The index of e is, blank space, double quotes, plus, index Of E, right parentheses, semicolon. Line 32, indented twice: System dot out dot print l n, left parentheses, double quotes, The index of l is, blank space, double quotes. Line 33, indented multiple times: plus, greeting, dot, index Of, left parentheses, single quote, l, single quote, right parentheses, right parentheses, semicolon. Line 34, indented twice: System dot out dot print l n, left parentheses, double quotes, The index of l o is, double quotes. Line 35, indented multiple times: plus, greeting dot index Of, left parentheses, double quotes, l o, double quotes, right parentheses, right parentheses, semicolon. Line 36, indented twice: System dot out dot print l n, left parentheses, double quotes, The index of h is, blank space, double quotes. Line 37, indented multiple times: plus, greeting dot index Of, left parentheses, single quote, h, single quote, right parentheses, right parentheses, semicolon. Line 38: blank. Line 39, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The midddle three characters of Hello are, blank space, double quotes. Line 40, indented multiple times: plus, greeting dot substring, left parentheses, 1, comma, 4, right parentheses, right parentheses, semicolon. Line 41, indented multiple times: System dot out dot print l n, left parentheses, double quotes, All characters of Hello except the first are, blank space, double quotes. Line 42, indented multiple times: plus, greeting dot substring, left parentheses, 1, right parentheses, right parentheses, semicolon. Line 43: blank. Line 44, indented twice: String s 4, equals, double quotes, backslash, n, backslash, t, tab space, backslash, t, tabspace, backslash, n, backslash, t, blank space, double quotes, semicolon. Line 45, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, Is s 4 blank, question mark, blank space, double quotes, plus, s 4 dot is Blank, left parentheses, right parenteses, right parentheses, semicolon. Line 46, indented twice: s 4, equals, s 4 dot strip, left parentheses, right parentheses, semicolon. Line 47, indented twice: System dot out dot print l n, left parentheses, double quotes, Is s 4 now empty, question mark, blank space, double quotes, plus, s 4 dot is Empty, left parentheses, right parentheses, right parentheses, semicolon. Line 48, indented once: right brace. Line 49: right brace.

Back to Example

System dot out dot print l n, left parentheses, double quotes, The sum of 1 and 2 is, double quotes, plus, left parentheses, 1 plus 2, right parentheses, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, int. Line 1: Forward slash, asterisk. Demonstrating string processing. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: public class String Processing. Line 5: left brace. Line 6, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 7, indented once: left brace. Line 8, indented twice: String inverted Name, equals, double quotes, lincoln, comma, abraham, double quotes, semicolon. Line 9: blank. Line 10, indented twice: int comma, equals, inverted Name, dot, index Of, left parentheses, single quote, comma, single quote, right parentheses, semicolon. 2 forward slashes, find the comma. Line 11, indented twice:

Back to Example

The table has 2 rows labelled Character and Index. The column entries are as follows.
Column 1: Character: l. Index: 0.
Column 2: Character: i. Index: 1.
Column 3: Character: n. Index: 2.
Column 4: Character: c. Index: 3.
Column 5: Character: o. Index: 4.
Column 6: Character: l. Index: 5.
Column 7: Character: n. Index: 6.
Column 8: Character: comma. Index: 7.
Column 9: Character: blank space. Index: 8.
Column 10: Character: a. Index: 9.
Column 11: Character: b. Index: 10.
Column 12: Character: r. Index: 11.
Column 13: Character: a. Index: 12.
Column 14: Character: h. Index: 13.
Column 15: Character: a. Index: 14.
Column 16: Character: m. Index: 15.

Back to Figure

The table has 2 rows labelled Character and Index. The column entries are as follows.
Column 1: Character: l. Index: 0.
Column 2: Character: i. Index: 1.
Column 3: Character: n. Index: 2.
Column 4: Character: c. Index: 3.
Column 5: Character: o. Index: 4.
Column 6: Character: l. Index: 5.
Column 7: Character: n. Index: 6.
Column 8: Character: comma. Index: 7.
Column 9: Character: blank space. Index: 8.
Column 10: Character: a. Index: 9.
Column 11: Character: b. Index: 10.
Column 12: Character: r. Index: 11.
Column 13: Character: a. Index: 12.
Column 14: Character: h. Index: 13.
Column 15: Character: a. Index: 14.
Column 16: Character: m. Index: 15. An arrow pointing to the character, comma, in index 7 reads: index Of Comma.

Back to Figure

The table has 2 rows labelled Character and Index. The column entries are as follows.
Column 1: Character: l. Index: 0.
Column 2: Character: i. Index: 1.
Column 3: Character: n. Index: 2.
Column 4: Character: c. Index: 3.
Column 5: Character: o. Index: 4.
Column 6: Character: l. Index: 5.

Column 7: Character: n. Index: 6.
Column 8: Character: comma. Index: 7.
Column 9: Character: blank space. Index: 8.
Column 10: Character: a. Index: 9.
Column 11: Character: b. Index: 10.
Column 12: Character: r. Index: 11.
Column 13: Character: a. Index: 12.
Column 14: Character: h. Index: 13.
Column 15: Character: a. Index: 14.
Column 16: Character: m. Index: 15. An arrow pointing to the character, comma, in index 7 reads: index Of Comma. A double headed arrow between indices 0 to 6 reads: extracted to last Name.

Back to Code

The table has 2 rows labelled Character and Index. The column entries are as follows.
Column 1: Character: l. Index: 0.
Column 2: Character: i. Index: 1.
Column 3: Character: n. Index: 2.
Column 4: Character: c. Index: 3.
Column 5: Character: o. Index: 4.
Column 6: Character: l. Index: 5.
Column 7: Character: n. Index: 6.
Column 8: Character: comma. Index: 7.
Column 9: Character: blank space. Index: 8.
Column 10: Character: a. Index: 9.
Column 11: Character: b. Index: 10.
Column 12: Character: r. Index: 11.

Column 13: Character: a. Index: 12.
Column 14: Character: h. Index: 13.
Column 15: Character: a. Index: 14.
Column 16: Character: m. Index: 15. An arrow pointing to the character, comma, in index 7 reads: index Of Comma. An arrow pointing to the character, a, in index 9 reads: index Of Comma plus 2. A double headed arrow between indices 9 to 15 reads: extracted to first Name.

Back to Figure

Each table has 2 rows labelled Character and Index.
The first table is labelled first Name. The column entries are as follows.
Column 1. Character: lowercase a. Index: 0.
Column 2. Character: lowercase b. Index: 1.
Column 3. Character: lowercase r. Index: 2.
Column 4. Character: lowercase a. Index: 3.
Column 5. Character: lowercase h. Index: 4.
Column 6. Character: lowercase a. Index: 5.
Column 7. Character: lowercase m. Index: 6.
The second table is labelled first Name Start. The column entries are as follows.
Column 1. Character: uppercase a. Index: 0.
An arrow is drawn from the first column of first Name to the first column of first Name start.
The third table is labelled first Name Remainder. The column entries are as follows.
Column 1. Character: lowercase b. Index: 1.
Column 2. Character: lowercase r. Index: 2.

Column 3. Character: lowercase a. Index: 3.
Column 4. Character: lowercase h. Index: 4.
Column 5. Character: lowercase a. Index: 5.
Column 6. Character: lowercase m. Index: 6.
An arrow is drawn from columns 2 to 7 of first Name to first name Remainder.
A plus sign is marked between first Name Start and first Name Remainder.
The fourth table is labelled first Name. The column entries are as follows.
Column 1. Character: uppercase a. Index: 0.
Column 2. Character: lowercase b. Index: 1.
Column 3. Character: lowercase r. Index: 2.
Column 4. Character: lowercase a. Index: 3.
Column 5. Character: lowercase h. Index: 4.
Column 6. Character: lowercase a. Index: 5.
Column 7. Character: lowercase m. Index: 6.
Arrows are drawn from first Name Start and first Name Remainder to first Name.

Back to Figure

The table has 2 columns labelled Package and Categories of Classes. The row entries are as follows.
Row 1. Package: java dot lang. Categories of Classes: Basic functionality common to many programs, such as the String class, Math class, and object wrappers for the primitive data types.
Row 2. Package: java dot text. Categories of Classes: Classes for formatting numeric output.

Row 3. Package: java dot util. Categories of Classes: The Scanner class, the Random class, and other miscellaneous classes.
Row 4. Package: java dot i o. Categories of Classes: Classes for reading from and writing to files.

Back to Table

import java dot text dot Decimal Format, semicolon.

Back to Code

The table has 2 columns labelled Package and Categories of Classes. The row entries are as follows.
Row 1. Package: java dot lang. Categories of Classes: Basic functionality common to many programs, such as the String class, Math class, and object wrappers for the primitive data types.
Row 2. Package: java dot text. Categories of Classes: Classes for formatting numeric output.
Row 3. Package: java dot util. Categories of Classes: The Scanner class, the Random class, and other miscellaneous classes.
Row 4. Package: java dot i o. Categories of Classes: Classes for reading from and writing to files.

Back to Table

The table has 2 columns labelled Package and Categories of Classes. The row entries are as follows.
Row 1. Package: java dot lang. Categories of Classes:

Basic functionality common to many programs, such as the String class, Math class, and object wrappers for the primitive data types.
Row 2. Package: java dot text. Categories of Classes: Classes for formatting numeric output.
Row 3. Package: java dot util. Categories of Classes: The Scanner class, the Random class, and other miscellaneous classes.
Row 4. Package: java dot i o. Categories of Classes: Classes for reading from and writing to files.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, String, double, int. Line 1: Forward slash, asterisk. Demonstrating the Decimal Format class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: 2 forward slash. import the Decimal Format class from the java dot text package. Line 6: import java dot text dot Decimal Format. Line 7: blank. Line 8: public class Demo Decimal Format. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: 2 forward slashes. first, comma, instantiate a Decimal Format object specifying a. Line 13, indented twice: 2 forward slashes. pattern for currency. Line 14, indented twice: Decimal Format price Pattern, comma, new Decimal Format, left parentheses, double quotes, dollar sign, 0.00, double quotes, right parentheses, semicolon. Line 15: blank. Line 16, indented twice: double price 1, equals, 78.66666666, semicolon. Line 17, indented twice: double price 2, equals, 34.5, semicolon. Line 18, indented twice: double price 3, equals, decimal point, 3333333, semicolon. Line 19, indented twice: int price 4, equals, 3, semicolon. Line 20, indented twice: double price 5, equals, 100.23, semicolon. Line 21: blank. Line 22, indented twice: 2 forward slashes, then print the values using the pattern. Line 23, indented twice: System dot out dot print l n, left parentheses, double quotes, The first price is, colon, blank space, double quotes. Line 24, indented multiple times: plus, price Pattern, dot, format, left parentheses, price 1, right parentheses, right parentheses, semicolon. Line 25, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash n, The second price is, colon, blank space, double quotes. Line 26, indented multiple times: plus, price Pattern dot format, left parentheses, price 2, right parentheses, right parentheses, semicolon. Line 27, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The third price is, colon, blank space, double quotes. Line 28, indented multiple times: plus, price Pattern, dot, format, left parentheses, price 3, right parentheses, right parentheses, semicolon. Line 29, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The fourth price is, colon, blank slash, double quotes. Line 30, indented multiple times: plus, price Pattern, dot, format, left parentheses, price 4, right parentheses, right parentheses, semicolon. Line 31, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The fifth price is, colon, blank space, double quotes. Line 32, indented multiple times: plus, price Pattern dot format, left parentheses, price 5, right parentheses, right parentheses, semicolon. Line 33: blank. Line 34, indented twice: 2 forward slashes, instantiate another new Decimal Format object. Line 35, indented twice: for printing percentages. Line 36, indented twice: Decimal Format percent pattern, equals, new Decimal Format, left parentheses, double quotes, 0.0, hash, percentage, double quotes, right parentheses, semicolon. Line 37: blank. Line 38, indented twice: double average 1, equals, decimal point, 98, semicolon. Line 39, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The first average is, colon, blank space, double quotes. Line 40, indented multiple times: plus, percent Pattern dot format, left parentheses, average 1, right parentheses, right parentheses, semicolon. Line 41, indented twice: double average 2, equals, decimal point, 98748, semicolon. Line 42, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The second average is, colon, blank space, double quotes. Line 43, indented multiple times: plus, percent Pattern dot format, left parentheses, average 2, right parentheses, right parentheses, semicolon. Line 44: blank. Line 45, indented twice: 2 forward slashes. now instantiate another new Decimal Format object. Line 46, indented twice: 2 forward slashes. for printing time as two digits. Line 47, indented twice: Decimal Format time Pattern, equals, new Decimal Format, left parentheses, double quotes, 0.0, double quotes, right parentheses, semicolon. Line 48: blank. Line 49, indented twice: int hours, equals, 5, comma, minutes, equals, 12, comma, seconds, equals, 0, semicolon. Line 50, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The time is, blank space, double quotes. Line 51, indented multiple times: plus, time Pattern dot format, left parentheses, hours, right parentheses, plus, double quotes, colon, double quotes. Line 52, indented multiple times, plus, time Pattern dot format, left parentheses, minutes, right parentheses, plus, double quotes, colon, double quotes. Line 53, indented multiple times: plus, time Pattern dot format, left parentheses, seconds, right parentheses, right parentheses, semicolon. Line 54: blank. Line 55, indented twice: 2 forward slashes, now instantiate another Decimal Format object. Line 56, indented twice: 2 forward slashes, for printing numbers in the millions. Line 57, indented twice: Decimal Format big Number, equals, new Decimal Format, left parentheses, double quotes, hash, comma, hash, hash, hash, double quotes, right parentheses, semicolon. Line 58: blank. Line 59, indented twice: int millions, equals, 1234567, semicolon. Line 60, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, millions is, blank space, double quotes. Line 61, indented multiple times: plus, big Number dot format, left parentheses, millions, right parentheses, right parentheses, semicolon. Line 62, indented once: right brace. Line 63: right brace.

Back to Example

Double quotes, dollar sign, 0.00, dollar sign.

Back to Code

The description of the Random class constructor is as follows. Random, left parentheses, right parentheses, creates a random number generator with a seed generated using the system time. The second table describes the next Int method. The table has 2 columns labelled Return Value and Method name and argument list. The row entry is as follows. Return value: int. Method name and argument list: next Int, left parentheses, int number, right parentheses, returns a random integer ranging from 0 up to, but not including, number in uniform distribution.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, String, int. Line 1: Forward slash, asterisk. A demonstration of the Random Class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: import java dot util dot Random, semicolon. Line 5: blank. Line 6:

public class Random Numbers. Line 7: left brace. Line 8, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: Random random, equals, new Random, left parentheses, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: 2 forward slashes. simulate the roll of a die. Line 13, indented twice: int die, equals, random dot next Int, left parentheses, 6, right parentheses, plus, 1, semicolon. Line 14, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The die roll is, blank space, double quotes, plus, die, right parentheses, semicolon. Line 15: blank. Line 16, indented once: 2 forward slashes, generate a random number between 20 and 200. Line 17, indented twice: int start, equals, 20, comma, end, equals, 200, semicolon. Line 18, indented twice: int number, equals, random dot next Int, left parentheses, end minus start plus 1, right parentheses, plus, start, semicolon. Line 19, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The random number between, blank space, double quotes, plus, start. Line 20, indented multiple times: plus, double quotes, blank space, and, blank space, double quotes, plus, end, plus, double quotes, blank space, is, blank space, double quotes, plus, number, right parentheses, semicolon. Line 21, indented once: right brace. Line 22: right brace.

Back to Example

The syntax of the Scanner class constructor is as follows. Scanner, left parentheses, Input Stream data Source, right parentheses. Creates a Scanner object that will read from the Input Stream data Source. To read from the keyboard, we will use the predefined Input Stream System dot in. The table is titled Selected Methods of the Scanner Class. The table has 2 columns labelled Return Value and Method Name and Argument List. The row entries are as follows.

Row 1. Return Value: byte. Method Name and Argument List: next Byte, left parentheses, right parentheses, returns the next input as a byte.

Row 2. Return Value: short. Method Name and Argument List: next Short, left parentheses, right parentheses, returns the next input as a short.

Row 3. Return Value: int. Method Name and Argument List: next Int, left parentheses, right parentheses, returns the next input as an int.

Row 4. Return Value: long. Method Name and Argument List: next Long, left parentheses, right parentheses, returns the next input as a long.

Row 5. Return Value: float. Method Name and Argument List: next Float, left parentheses, right parentheses, returns the next input as a float.

Row 6. Return Value: double. Method Name and Argument List: next Double, left parentheses, right parentheses, returns the next input as a double.

Row 7. Return Value: boolean. Method Name and Argument List: next Boolean, left parentheses, right

parentheses, returns the next input as a boolean.
Row 8. Return Value: string. Method Name and Argument List: next, left parentheses, right parentheses, returns the next token in the input line as a String.
Row 9. Return Value: string. Method Name and Argument List: next Line, left parentheses, right parentheses, returns the unread characters of the input line as a String.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, String, int, float. Line 1: Forward slash, asterisk. A demonstration of reading from the keyboard using Scanner. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Data Input. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 12: blank. Line 13, indented twice: System dot out dot print, left parentheses, double quotes, Enter your first name, right angled bracket, double quotes, right parentheses, semicolon. Line 14, indented twice: String first Name, equals, scan dot next, left parentheses, right parentheses, semicolon. Line 15, indented twice: System dot out dot print l n, left parentheses, double quotes, Your name is, blank space, double quotes, plus, first Name, right paentheses, semicolon. Line 16: blank. Line 17, indented twice: System dot out dot print, left parentheses, double quotes, backslash, n, Enter your age as an integer, right angled bracket, double quotes, right parentheses, semicolon. Line 18, indented twice: int age, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 19, indented twice: System dot out dot print l n, left parentheses, double quotes, Your age is, blank space, double quotes, plus, age, right parentheses, semicolon. Line 20: blank. Line 21, indented twice: System dot out dot print, left parentheses, double quotes, back slash, n, Enter your G P A, right angled bracket, double quotes, right parentheses, semicolon. Line 22, indented twice: float g p a, equals, scan dot next Float, left parentheses, right parentheses, semicolon. Line 23, indented twice: System dot out dot print l n, left parentheses, double quotes, Your G P A is, blank space, double quotes, plus, g p a, right parentheses, semicolon. Line 24, indented once: right brace. Line 25: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, new, char. Line 1: Forward slash, asterisk. A demonstration of how to get character input using Scanner. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class

Character Input. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 12: blank. Line 13, indented twice: System dot out dot print, left parentheses, double quotes, Enter your middle initial, right angled bracket, blank space, double quotes, right parentheses, semicolon. Line 14, indented twice: String initial S, equals, scan dot next, left parentheses, right parentheses, semicolon. Line 15, indented twice: char initial, equals, initial S dot char At, left parentheses, 0, right parentheses, semicolon. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, Your middle initial is, blank space, double quotes, plus, initial, right parentheses, semicolon. Line 17, indented once: right brace. Line 18: right brace.

Back to Example

The table has 2 columns labelled Character and Unicode equivalents. The row entries are as follows.
Row 1. Character: space. Unicode equivalents: back slash, u 0 0 A 0, comma, back slash, u 2 0 0 7, comma, back slash, u 2 0 2 F.
Row 2. Character: tab. Unicode equivalents: back slash, u 0 0 0 9, comma, back slash, u 0 0 0 B.
Row 3. Character: line feed. Unicode equivalents: back

slash, u 0 0 0 A.
Row 4. Character: form feed. Unicode equivalents: back slash, u 0 0 0 C.
Row 5. Character: carriage return. Unicode equivalents: back slash, u 0 0 0 D.
Row 6. Character: file, group, unit, and record separators. Unicode equivalents: back slash, u 0 0 1 C, comma, back slash, u 0 0 1 D, comma, back slash, u 0 0 1 E, comma, back slash, u 0 0 1 F.

Back to Table

Left angled bracket, tab, right angled bracket, Jon, left angled bracket, space, right angled bracket, Olsen, comma, left angled bracket, space, right angled bracket, 21, left angled bracket, space, right angled bracket.

Back to Code

Left angled bracket, space, right angled bracket, left angled bracket, space, right angled bracket, left angled bracket, space, right angled bracket, left angled bracket, tab, right angled bracket, 21, left angled bracket, newline, right angled bracket.

Back to Code

Left angled bracket, tab, right angled bracket, Jon, left angled bracket, space, right angled bracket, Olsen, comma, left angled bracket, space, right angled bracket, 21, left angled bracket, space, right angled bracket.

Back to Code

Program code. The code uses the following keywords: import, public, class, static, void, String. Line 1: Forward slash, asterisk. A demonstration of using Scanner's next Line method. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Input A Line. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 12: blank. Line 13, indented twice: System dot out dot print, left parentheses, double quotes, Enter a sentence, right angled bracket, double quotes, right parentheses, semicolon. Line 14, indented once: String sentence, equals, scan dot next Line, left parentheses, right parentheses, semicolon. Line 15, indented once: System dot out dot print l n, left parentheses, double quotes, You said, colon, back slash, double quotes, double quotes, plus, sentence, plus, double quotes, backslash, double quotes, double quotes, right parentheses, semicolon. Line 16, indented once: right brace. Line 17: right brace.

Back to Example

The first table has 2 columns labelled Constant and value.

The row entries are as follows.

Row 1. Constant: in. Value: Static constant that represents the standard input stream, by default the keyboard.

Row 2. Constant: out. Value: static constant that represents the standard output stream, by default the Java console.

The second table has 2 columns labelled Return value and Method name and argument list. The row entries are as follows.

Row 1. Return value: void. Method name and argument list: exit, left parentheses, int exit Status, right parentheses. Static method that terminates the Java Virtual Machine. A value of 0 for exit Status indicates a normal termination. Any other values indicate an abnormal termination and are used to signal that the program ended because an error occurred.

Back to Table

The table has 2 columns labelled Return value and Method name and argument list. The row entries are as follows.

Row 1. Return value: void. Method name and argument list: print, left parentheses, argument, right parentheses, prints argument to the standard output device. The argument can be any primitive data type, a String object, or another object reference.

Row 2. Return value: void. Method name and argument list: print l n, left parentheses, argument, right

parentheses, prints argument to the standard output device, then prints a newline character. The argument can be any primitive data type, a String, or another object reference.
Row 3. Return value: void. Method name and argument list: print l n, left parentheses, right parentheses, prints a newline character. This method is useful for skipping a line in the program's output.

Back to Table

Program code. The code contains the following keywords: public, class, static, void, String. Line 1: Forward slash, asterisk. Testing the print and print l n methods. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Print Demo. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: System dot out dot print l n, left parentheses, double quotes, Combine the arguments using concatenation, double quotes, right parentheses, semicolon. Line 10, indented twice: System dot out dot print l n, left parentheses, double quotes, A double, colon, blank space, double quotes, plus, 23.7, plus, double quotes, comma, and an int, colon, blank space, double quotes, plus, 78, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: System dot out dot print, left parentheses, double quotes, backslash, n, Java is case sensitive, colon, blank space, double quotes, right parentheses, semicolon. Line 13, indented twice: System dot out dot print l n, left parentheses, single quote, a, single quote, plus, double quotes, blank space, is different from, blank space, double quotes, plus, single quote, uppercase a, single quote, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, Create a variable and print its value, double quotes, right parentheses, semicolon. Line 16, indented twice: String s, equals, new String, left parentheses, double quotes, The grade is, double quotes, right parentheses, semicolon. Line 17, indented twice: double grade, equals, 3.81, semicolon. Line 18, indented twice: System dot out dot print l n, left parentheses, s, plus, double quotes, blank space, double quotes, plus, grade, right parentheses, semicolon. Line 19: blank. Line 20, indented twice: System dot exit, left parentheses, 0, right parentheses, semicolon. 2 forward slashes. optional. Line 21, indented once: right brace. Line 22: right brace.

Back to Example

The table has 2 columns labelled Constant and Value. The row entries are as follows.
Row 1: Constant: uppercase e. Value: lowercase e, comma, the base of the natural logarithm.
Row 2: Constant: uppercase PI. Value: pi, comma, the ratio of the circumference of a circle to its diameter.

Back to Table

Line 1: Math dot E. Line 2: Math dot PI.

Back to Code

The table has 2 columns labelled Return value and Method name and argument list. The row entries are as follows.
Row 1. Return value: data Type Of Arg. Method name and argument list: a b s, left parentheses, a r g, right parentheses, static method that returns the absolute value of the argument a r g, which can be a double, float, int, or long.
Row 2. Return value: double. Method name and argument list: log, left parentheses, double a r g, right parentheses, static method that returns the natural logarithm, left parentheses, in base e, right parentheses, of its argument, a r g. For example, log, left parentheses, 1, right parentheses, returns 0 and log, left parentheses, Math dot uppercase e, right parentheses returns 1.
Row 3. Return value: data Type Of A r g s. Method name and argument list: min, left parentheses, a r g uppercase a, comma, a r g uppercase b, right parentheses, static method that returns the smaller of the two arguments. The arguments can be doubles, floats, ints or longs.
Row 4. Return value: data Type Of A r g s. Method name and argument list: max, left parentheses, a r g uppercase a, comma, a r g uppercase b, right parentheses, static method that returns the larger of the two arguments. The

arguments can be doubles, floats, ints, or longs.
Row 5. Return value: double. Method name and argument list: p o w, left parentheses, double base, comma, double e x p, right parentheses, static method that returns the value of base raised to the e x p power.
Row 6. Return value: long. Method name and argument list: round, left parentheses, double a r g, right parentheses, static method that returns the closest integer to its argument, a r g.
Row 7. Return value: double. Method name and argument list: s q r t, left parentheses, double a r g, right parentheses, static method that returns the positive square root of a r g.

Back to Table

Math dot a b s, left parentheses, negative 5, right parentheses.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, double. Line 1: Forward slash, asterisk. A demonstration of the Math class methods and constants. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Math Constants. Line 6: left brace Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once:

left brace. Line 9, indented twice: System dot out dot print l n, left parentheses, double quotes, The value of e is, blank space, double quotes, plus, Math dot E, right parentheses, semicolon. Line 11, indented twice: System dot out dot print l n, left parentheses, double quotes, The value of pi is, blank space, double quotes, plus, Math dot PI, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: double d 1, equals, Math dot a b s, left parentheses, 6.7, right parentheses, semicolon. 2 forward slashes, d 1 will be assigned 6.7. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The absolute value of 6.7 is, blank space, double quotes, plus, d 1, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: double d 2, equals, Math dot a b s, left parentheses, negative 6.7, right parentheses, semicolon. 2 forward slashes, d 2 will be assigned 6.7. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The absolute value of negative 6.7 is, blank space, double quotes, plus, d 2, right parentheses, semicolon. Line 17, indented once: right brace. Line 18: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, string, double. Line 1: Forward slash, asterisk. A demonstration of some Math class methods. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: public class Math Methods. Line 6: left brace. Line

7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: double d 2, equals, Math dot log, left parentheses, 5, right parentheses, semicolon. Line 10, indented once: System dot out dot print l n, left parentheses, backslash, n, The log of 5 is, blank space, double quotes, plus, d 2, right parentheses, semicolon. Line 12, indented twice: double d 4, equals, Math dot s q r t, left parentheses, 9, right parentheses, semicolon. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The square root of 9 is, blank space, double quotes, plus, d 4, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: double four Cubed, equals, Math dot p o w, left parentheses, 4, comma, 3, right parentheses, semicolon. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, 4 to the power 3 is, blank space, double quotes, plus, four Cubed, right parentheses, semicolon. Line 17: blank. Line 18, indented twice: double big Number, equals, Math dot p o w, left parentheses, 43.5, comma, 3.4, right parentheses, semicolon. Line 19, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, 43.5 to the power 3.4 is, blank space, double quotes, plus, big Number, right parentheses, semicolon. Line 20, indented once: right brace. Line 21: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String. Line 1: Forward slash, asterisk. A demonstration of the Math round method. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: public class Math Rounding. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, Sting, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: System dot out dot print l n, left parentheses, double quotes, 23.4 rounded is, blank space, double quotes, plus, Math dot round, left parentheses, 23.4, right parentheses, right parentheses, semicolon. Line 10, indented once: System dot out dot print l n, left parentheses, double quotes, 23.49 rounded is, blank space, double quotes, plus, Math dot round, left parentheses, 23.49, right parentheses, right parentheses, semicolon. Line 11, indented twice: System dot out dot print l n, left parentheses, double quotes, 23.5 rounded is, blank space, double quotes, plus, Math dot round, left parentheses, 23.5, right parentheses, right parentheses, semicolon. Line 12, indented twice: System dot out dot print l n, left parentheses, double quotes, 23.51 rounded is, blank space, double quotes, plus, Math dot round, left parentheses, 23.51, right parentheses, right parentheses, semicolon. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, 23.6 rounded is, blank space, double quotes, plus, Math dot round, left parentheses, 23.6, right parentheses, right parentheses, semicolon. Line 14, indented once: right brace. Line 15: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, int. Line 1: Forward slash, asterisk. A demonstration of min and max Math class methods. Line 2, indented once: Anderson, comma, franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: public class Math Min Max Methods. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: int smaller, equals, Math dot min, left parentheses, 8, comma, 2, right parentheses, semicolon. Line 10, indented twice: System dot out dot print l n, left parentheses, double quotes, The smaller of 8 and 2 is, blank space, double quotes, plus, smaller, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: int larger, equals, Math dot max, left parentheses, 8, comma, 2, right parentheses, semicolon. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, The larger of 8 and 2 is, blank space, double quotes, plus, larger, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: int a equals 8, comma, b equals 5, comma, c equals 12, semicolon. Line 16, indented twice: int temp Smaller, equals, Math dot min, left parentheses, a, comma, b, right parentheses, semicolon. 2 forward slashes, find smaller of a ampersand b. Line 17, indented twice: int smallest, equals, Math dot min, left parentheses, temp Smaller, comma, c, right parentheses, semicolon. 2 forward slashes, compare result to c. Line 18, indented twice: System dot out dot print l n, left parentheses, double quotes, The smallest of, blank space, double quotes, plus, a, plus, double quotes, comma, blank space, double quotes, plus, b, plus, double quotes, comma, and, blank space, double quotes. Line 19, indented multiple times: plus, c, plus, double quotes, blank space, is, blank space, double quotes, plus, smallest, right parentheses, semicolon. Line 20, indented once: right brace. Line 21: right brace.

Back to Example

import java dot text dot Number Format, semicolon.

Back to Code

The table has 2 columns labelled Return Value and Method Name and Argument List. The row entries are as follows.
Row 1. Return Value: Number Format. Method name and argument list: get Currency Instance, left parentheses, right parentheses.
Row 2. Return Value: Number Format. Method name and argument list: get Percent Instance, left parentheses, right parentheses, static method that creates a format object for percentages.

Row 3. Return Value: String. Method name and argument list: format, left parentheses, double number, right parentheses, returns a String representation of number formatted as a currency or percentage, depending on the object used to call the method.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, double, String. Line 1: Forward slash, asterisk. Demonstration of currency and percentage formatting. Line 2, indented once: using the Number Format class. Line 3, indented once: Anderson, comma, Franceschi. Line 4: asterisk, forward slash. Line 5: blank. Line 6: 2 forward slashes. we need to import the Number Format class from java dot text. Line 7: import java dot text dot Number Format, semicolon. Line 8: blank. Line 9: public class Demo Number Format. Line 10: left brace. Line 11, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 12, indented once: left brace. Line 13, indented twice: double winning Percentage, equals, decimal point, 675, semcolon. Line 14, indented twice: double price, equals, 78.9, semicolon. Line 15: blank. Line 16, indented twice: 2 forward slashes. get a Number Format object for printing a percentage. Line 17, indented twice: Number Format percent Format, equals, Number Format, dot, get Percentage Instance, left parentheses, right parentheses, semicolon. Line 18: blank. Line 19, indented twice: 2 forward slashes. call format method using the Number Format object. Line 20, indented twice: System dot out dot print, left parentheses, double quotes, The winning percentage is, blank space, double quotes, right parentheses, semicolon. Line 21, indented twice: System dot out dot print l n, left parentheses, percent Format dot format, left parentheses, winning Percentage, right parentheses, right parentheses, semicolon. Line 22: blank. Line 23, indented twice: 2 forward slashes. get a Number Format object for printing currency. Line 24, indented twice: Number Format price Format, equals, Number Format dot get Currency Instance, left parentheses, right parentheses, semicolon. Line 25: blank. Line 26, indented twice: 2 forward slashes. call format method using the Number Format object. Line 27, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The price is, colon, blank space, double quotes. Line 28, indented multiple times: plus, price Format dot format, left parentheses, price, right parentheses, right parentheses, semicolon. Line 29, indented once: right brace. Line 30: right brace.

Back to Example

The row entries are as follows.
Row 1. Primitive Data Type: double. Wrapper Class: Double.
Row 2. Primitive Data Type: float. Wrapper Class: Float.
Row 3. Primitive Data Type: long. Wrapper Class: Long.
Row 4. Primitive Data Type: int. Wrapper Class: Integer.

Row 5. Primitive Data Type: short. Wrapper Class: Short.
Row 6. Primitive Data Type: byte. Wrapper Class: Byte.
Row 7. Primitive Data Type: char. Wrapper Class: Character.
Row 8. Primitive Data Type: boolean. Wrapper Class: Boolean.

Back to Table

Program code. The code contains the following keywords: public, class, static, void, String, int. Line 1: Forward slash, asterisk. A demonstration of the Wrapper classes and methods. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Demo Wrapper. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: int int Primitive, equals, 42, semicolon. Line 10, indented twice: Integer integer Object, equals, int Primitive, semicolon. Line 11: blank. Line 12, indented twice: System dot out dot print l n, left parentheses, double quotes, The int is, blank space, double quotes, plus, int Primitive, right parentheses, semicolon. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, The Integer object is, blank space, double quotes, plus, integer Object, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: int sum, equals, int Primitive, plus, integer Object, semicolon. Line 16, indented twice:

System dot out dot print l n, left parentheses, double quotes, Their sum is, blank space, double quotes, plus, sum, right parentheses, semicolon. Line 17: blank. Line 18, indented twice: int i 1, equals, Integer dot parse Int, left parentheses, double quotes, 76, double quotes, right parentheses, semicolon. 2 forward slashes, convert double quotes, 76, double quotes, to an int. Line 19, indented twice: Integer i 2, equals, Integer dot value Of, left parentheses, double quotes, 76, double quotes, right parentheses, semicolon. 2 forward slashes, convert, double quotes, 76, double quotes, to Integer. Line 20, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The value of i 1 is, blank space, double quotes, plus, i 1, right parentheses, semicolon. Line 21, indented twice: System dot out dot print l n, left parentheses, double quotes, The value of i 2 is, blank space, double quotes, plus, i 2, right parentheses, semicolon. Line 22: blank. Line 23, indented twice: double d 1, equals, Double dot parse Double, left parentheses, double quotes, 58.32, double quotes, right parentheses, semicolon. Line 24, indented twice: Double d 2, equals, Double dot value Of, left parentheses, double quotes, 58.32, double quotes, right parentheses, semicolon. Line 25, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash n, The value of d 1 is, blank space, double quotes, plus, d 1, right parentheses, semicolon. Line 26, indented twice: System dot out dot print l n, left parentheses, double quotes, The value of d 2 is, blank space, double quotes, plus, d 2, right parentheses, semicolon. Line 27, indented once: right brace. Line 28: right brace.

Back to Example

The first table is titled Useful methods of the Integer Wrapper Class. The table has 2 columns labelled Return value and Method name and argument list. The row entries are as follows.
Row 1. Return value: int. Method name and argument list: parse Int, left parentheses, String s, right parentheses, static method that converts the String s to an int and returns that value.
Row 2. Return value: Integer. Method name and argument list: value Of, left parentheses, String s, right parentheses, static method that converts the String s to an Integer object and returns that object.
The second table is titled Useful methods of the Double wrapper class. The table has 2 columns labelled Return value and Method name and argument list. The row entries are as follows.
Row 1. Return value: double. Method name and argument list: parse Double, left parentheses, String s, right parentheses, static method that converts the String s to a double and returns that value.
Row 2. Return value: Double. Method name and argument list: value Of, left parentheses, String s, right parentheses, static method that converts the String s to a Double object and returns that object.

Back to Table

The table is titled useful static methods of the character wrapper class. The table has 2 columns labelled Return value and Method name and argument list. The row entries are as follows.
Row 1. Return value: boolean. Method name and argument list: is Digit, left parentheses, char c, right parentheses, returns true if c is a character from, single quote, 0, single quote, to single quote, 9, single quote, or a digit in other languages; false, otherwise.
Row 2. Return value: boolean. Method name and argument list: is Letter, left parentheses, char c, right parentheses, returns true if c is a letter; false, otherwise.
Row 3. Return value: boolean. Method name and argument list: is Lower Case, left parentheses, char c, right parentheses, returns true if c is a lowercase letter; false, otherwise.
Row 4. Return value: boolean. Method name and argument list: is Upper Case, left parentheses, char c, right parentheses, returns true if c is an uppercase letter; false, otherwise.
Row 5. Return value: char. Method name and argument list: to Lower Case, left parentheses, char c, right parentheses, returns the lowercase version of c if c is a letter; otherwise it returns c unchanged.
Row 6. Return value: char. Method name and argument list: to Upper Case, left parentheses, char c, right parentheses, returns the uppercase version of c if c is a

letter; otherwise it returns c unchanged.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, String, char, int. Line 1: Forward slash, asterisk. Character Methods. Line 2, indented twice: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: import java dot util dot Scanner, semicolon. Line 5: blank. Line 6: public class Character Methods. Line 7: left brace. Line 8, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: System dot out dot print, left parentheses, double quotes, Enter a character, right angled bracket, double quotes, right parentheses, semicolon. Line 13, indented twice: String input, equals, scan.next, left parentheses, right parentheses, semicolon. Line 14, indented twice: char c, equals, input dot char At, left parentheses, 0, right parentheses, semicolon. Line 15: blank. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, The Unicode decimal value of, blank space, double quotes, plus, c, plus, double quotes, blank space, is, colon, blank space, double quotes. Line 17, indented multiple times: plus, left parentheses, int, right parentheses, c, left parentheses, semicolon. Line 18:

blank. Line 19, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, double quotes, plus, c, plus, double quotes, blank space, is a digit, colon, blank space, double quotes. Line 20, indented multiple times: plus, Character dot is Digit, left parentheses, c, right parentheses, right parentheses, semicolon. Line 21: blank. Line 22, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash n, double quotes, plus, c, plus, double quotes, blank space, is a letter, colon, blank space, double quotes. Line 23, indented multiple times: plus, Character dot is Letter, left parentheses, c, right parentheses, right parentheses, semicolon. Line 24, indented twice: System dot out dot print l n, left parentheses, c, plus, double quotes, blank space, is uppercase, colon, blank space, double quotes. Line 25, indented multiple times: plus, Character dot is Upper Case, left parentheses, c, right parentheses, right parentheses, semicolon. Line 26, indented twice: System dot out dot print l n, left parentheses, c, plus, double quotes, blank space, is lowercase, colon, blank space, double quotes. Line 27, indented multiple times: plus, Character dot is Lower Case, left parentheses, c, right parentheses, right parentheses, semicolon. Line 28: blank. Line 29, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash n, double quotes, plus, c, plus, double quotes, blank space, in uppercase is, colon, blank space, double quotes. Line 30, indented multiple times: plus, Character dot to Upper

Case, left parentheses, c, right parentheses, right parentheses, semicolon. Line 31, indented twice: System dot out dot print l n, left parentheses, c, plus, double quotes, blank space, in lowercase is, colon, blank space, double quotes. Line 32, indented multiple times: plus, Character dot to Lower Case, left parentheses, c, right parentheses, right parentheses, semicolon. Line 33, indented once: right brace. Line 34: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String. Line 1: Forward slash, asterisk. Chapter 3 Programming Activity 2. Line 2, indented twice: Calling class methods. Line 3, indented twice: Anderson, comma, Franceschi. Line 4: Asterisk, forward slash. Line 5: blank. Line 6: 2 forward slashes. 5 asterisks. 1. add your import statements here. Line 7: blank. Line 8: public class Practice Methods. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11, indented twice: left brace. Line 12, indented twice: 2 forward slashes. 5 asterisks. Line 13, indented twice: 2 forward slashes. 2. a. Create a Scanner object to read from the keyboard. Line 14, indented twice: 2 forward slashes. Tab space. b. Prompt the user for their first name. Line 15, indented twice: 2 forward slashes. Tab space. c. Print a message that says hello to the user. Line 16, indented twice: 2 forward slashes. Tab space. d. Print a message that says how many letters. Line 17, indented once: 2 forward slashes. Tab spaces. are in the user's name. Line 18, indented once: 2 forward slashes. Your code goes here. Line 19: blank. Line 20: blank. Line 21, indented twice: 2 forward slashes. 5 asterisks. Line 22, indented twice: 2 forward slashes. 3. a. Skip a line, comma, then prompt the user for the year. Line 23, indented twice: 2 forward slashes. Tab spaces. they were born. Line 24, indented twice: 2 forward slashes. Tab spaces. b. Declare a constant for the current year. Line 25, indented twice: 2 forward slashes. Tab spaces. c. Calculate and print the age the user will be this year. Line 26, indented twice: 2 forward slashes. Tab spaces. d. Declare a constant for average life expectancy, comma. Line 27, indented twice: 2 forward slashes. Tab spaces. set its value to 78.94. Line 28, indented twice: 2 forward slashes. Tab spaces. e. Calculate and print the percentage. Line 29, indented twice: 2 forward slashes. Tab spaces. of the user's expected life they've lived. Line 30, indented twice: 2 forward slashes. Tab spaces. Use the Decimal Format class to format the percentage. Line 31, indented twice: 2 forward slashes. Tab spaces. to one decimal place. Line 32, indented twice: 2 forward slashes. Your code goes here. Line 33: blank. Line 34: blank. Line 35, indented twice: 2 forward slashes. 5 asterisks. Line 36, indented twice: 2 forward slashes. 4. a. Generate a secret random integer betwen 1 and 20. Line 37, indented twice: 2 forward slashes. Tab spaces. b. Skip a line, then ask the user for a guess. Line 38, indented twice: 2 forward slashes. Tab spaces. c. Print a message telling the user the secret number. Line 39, indented twice: 2 forward slashes. Tab spaces. and how far from the number the user's guess was. Line 40, indented twice: 2 forward slashes. Tab spaces. left parentheses, hint, colon, use Math dot a b s, right parentheses. Line 41, indented twice: 2 forward slashes. Your code goes here. Line 42: blank. Line 43, indented once: right brace. Line 44: right brace.

Back to Example

public static double foo, left parentheses, float f, right parentheses.

Back to Code

public static void foo, left parentheses, right parentheses.

Back to Code

public double foo, left parentheses, int i, comma, String s, comma, char c, right parentheses.

Back to Code

public double foo, left parentheses, int i, comma, String s, comma, char c, right parentheses.

Back to Code

Line 1: String s equals new String, left parentheses, double quotes, HI in uppercase, double quotes, right

parentheses, semicolon.
Line 2: System dot out dot print l n, left parentheses, s, right parentheses, semicolon.

Back to Code

Line 1: String s equals double quotes, uppercase a, double quotes, plus, double quotes, uppercase b uppercase c, double quotes, plus, double quotes, uppercase d uppercase e uppercase f, double quotes, plus, double quotes, uppercase g uppercase h uppercase i uppercase j, double quotes, semicolon.
Line 2: System dot out dot print l n, left parentheses, s, right parentheses, semicolon.

Back to Code

Line 1: String s equals double quotes, Hello, double quotes, semicolon.
Line 2: s equals s dot to Lower Case, left parentheses, right parentheses, semicolon.
Line 3: System dot out dot print l n, left parentheses, s, right parentheses, semicolon.

Back to Code

Line 1: int a equals Math dot min, left parentheses, 5, comma, 8, right parentheses, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

System dot out dot print l n, left parentheses, Math dot s q r t, left parentheses, 4.0, right parentheses, right parentheses, semicolon.

Back to Code

System dot out dot print l n, left parentheses, Math dot PI, right parentheses, semicolon.

Back to Code

Line 1: double f equals 5.7, semicolon.
Line 2: long i equals Math dot round, left parentheses, f, right parentheses, semicolon.
Line 3: System dot out dot print l n, left parentheses, i, right parentheses, semicolon.

Back to Code

System dot out dot print, left parentheses, Math dot round, left parentheses, 3.5, right parentheses, right parentheses, semicolon.

Back to Code

Line 1: int i equals Math dot a b s, left parentheses, negative 8, right parentheses, semicolon.
Line 2: System dot out dot print l n, left parentheses, i, right parentheses, semicolon.

Back to Code

Line 1: double d equals Math dot p o w, left parentheses, 2, comma, 3, right parentheses, semicolon.
Line 2: System dot out dot print l n, left parentheses, d, right parentheses, semicolon.

Back to Code

Line 1: String s 1 equals, double quotes, Intro, blank space, double quotes, semicolon.
Line 2: String s 2 equals, double quotes, to, double quotes, semicolon.
Line 3: String s 3 equals, double quotes, blank space, Programming, double quotes, semicolon.
Line 4: 2 forward slashes, your code goes here.

Back to Code

Line 1: String s equals, double quotes, Hello World, double quotes, semicolon.
Line 2: 2 forward slashes, your code goes here.

Back to Code

2 forward slashes, your code goes here.

Back to Code

Line 1: String s 1 equals double quotes, Welcome, blank space, double quotes, semicolon.
Line 2: String s 2 equals double quotes, to, blank space,

double quotes, semicolon.
Line 3: String s 3 equals double quotes, Java, blank space, double quotes, semicolon.
Line 4: String s 4 equals double quotes, Illuminated, double quotes, semicolon.
Line 5: 2 forward slashes, your code goes here.

Back to Code

2 forward slashes, your code goes here.

Back to Code

Line 1: int max, semicolon.
Line 2: 2 forward slashes, your code goes here.

Back to Code

Line 1: double d equals 5.0, semicolon.
Line 2: 2 forward slashes, your code goes here.

Back to Code

2 forward slashes, your code goes here.

Back to Code

2 forward slashes, your code goes here.

Back to Code

2 forward slashes, your code goes here.

Back to Code

2 forward slashes, your code goes here.

Back to Code

Line 1: String s equals double quotes, Hello World, double quotes, semicolon.
Line 2: System dot out dot print l n, left parentheses, s, right parentheses, semicolon.

Back to Code

Line 1: String s equals String, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon.
Line 2: System dot out dot print l n, left parentheses, s, right parentheses, semicolon.

Back to Code

Line 1: String s 1 equals double quotes, Hello, double quotes, semicolon. In the string Hello, character h is in uppercase.
Line 2: String s 2 equals double quotes, e l l o, double quotes, semicolon.
Line 3: String s equals s 1 minus s 2, semicolon.

Back to Code

Line 1: short s equals Math dot round, left parentheses, 3.2, right parentheses, semicolon.

Line 2: System dot out dot print l n, left parentheses, s, right parentheses, semicolon.

Back to Code

Line 1: int a equals Math dot p o w, left parentheses, 3, comma, 4, right parentheses, semicolon.
Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: double pi equals Math, left parentheses, uppercase p uppercase i, right parentheses, semicolon.
Line 2: System dot out dot print l n, left parentheses, pi, right parentheses, semicolon.

Back to Code

Line 1: String s equals, single quote, uppercase h, single quote, semicolon.
Line 2: System dot out dot print l n, left parentheses, double quotes, s is, blank space, double quotes, plus s, right parentheses, semicolon.

Back to Code

Program code. The words in the variable names are merged. The code contains the following keywords: public, class, static, void, main, String, int.
Line 1: public class Test.
Line 2: left brace.

Line 3, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses.
Line 4, indented once: left brace.
Line 5, indented twice: int a equals 6, semicolon.
Line 6, indented twice: Number Format n f equals Number Format dot get Currency Instance, left parentheses, right parentheses, semicolon.
Line 7, indented once: right brace.
Line 8: right brace.

Back to Code

The compiler message is as follows.
Line 1: Test dot java, colon, 6, colon, error, colon, cannot find symbol.
Line 2, indented once: Number Format n f equals Number Format dot get Currency Instance, left parentheses, right parentheses, semicolon.
Line 3: a caret sign below uppercase N of the first occurrence of Number Format in Line 2.
Line 4, indented once: symbol, colon, class Number Format.
Line 5, indented once: location, colon, class Test.
Line 6: Test dot java, colon, 6, colon, error, colon, cannot find symbol.
Line 7, indented once: Number Format n f equals Number Format dot get Currency Instance, left parentheses, right parentheses, semicolon.
Line 8: a caret sign below uppercase N of the second

occurrence of Number Format in Line 7.
Line 9, indented once: symbol, colon, variable Number Format.
Line 10, indented once: location, colon, class Test.
Line 11: 2 errors.

Back to Code

The code sequence has 3 lines which are as follows.
Line 1: String s, semicolon. 2 forward slashes, line 10.
Line 2: int l equals s dot length, left parentheses, right parentheses, semicolon. 2 forward slashes, line 11.
Line 3: System dot out dot print l n, left parentheses, double quotes, length is, blank space, double quotes, plus, l, right parentheses, semicolon. 2 forward slashes, line 12.

Back to Code

The compiler message is as follows.
Line 1: Test dot java, colon, 11, colon, error, colon, variable s might not have been initialized.
Line 2, indented once: int l equals s dot length left parentheses, right parentheses, semicolon, 2 forward slashes, line 11.
Line 3: a caret sign below lowercase s in Line 2.
Line 4: 1 error.

Back to Code

The code sequence has 2 lines which are as follows.

Line 1: double d equals math dot s q r t, left parentheses, 6, right parentheses, semicolon, 2 forward slashes, line 10.
Line 2: System dot out dot print l n, left parentheses, double quotes, d equals, double quotes, plus d, right parentheses, semicolon, 2 forward slashes, line 11.

Back to Code

The compiler message is as follows.
Line 1: Test dot java, colon, 10, colon, error, colon, cannot find symbol.
Line 2, indented once: double d equals math dot s q r t, left parentheses, 6, right parentheses, semicolon, 2 forward slashes, line 10.
Line 3: a caret sign below lowercase m of Line 2.
Line 4, indented once: symbol, colon, variable math.
Line 5, indented once: location, colon, class Test.
Line 6: 1 error.

Back to Code

The code sequence has 2 lines which are as follows.
Line 1: double d equals Math dot PI, left parentheses, right parentheses, semicolon. 2 forward slashes, line 10.
Line 2: System dot out dot print l n, left parentheses, double quotes, d equals double quotes, plus d, right parentheses, semicolon. 2 forward slashes, line 11.

Back to Code

The compiler message is as follows.
Line 1: Test dot java, colon, 10, colon, error, colon, cannot find symbol.
Line 2, indented once: double d equals Math dot PI, left parentheses, right parentheses, semicolon. 2 forward slashes, line 10.
Line 3: a caret sign below the dot in Line 2.
Line 4, indented once: symbol, colon, method PI, left parentheses, right parentheses.
Line 5, indented once: location, colon, class Math.
Line 6: 1 error.

Back to Code

The code sequence has 2 lines which are as follows.
Line 1: double d equals Math dot e, semicolon. 2 forward slashes, line 10
Line 2: System dot out dot print l n, left parentheses, double quotes, d, equals, double quotes, plus d, right parentheses, semicolon. 2 forward slashes, line 11.

Back to Code

The compiler message is as follows.
Line 1: Test dot java, colon, 10, colon, error, colon, cannot find symbol.
Line 2, indented once: double d equals Math dot e, semicolon.
Line 3: a caret sign below dot in Line 2.
Line 4, indented once: symbol, colon, variable e.

Line 5, indented once: location, colon, class Math.
Line 6: 1 error.

Back to Code

The code sequence has 3 statements which are as follows.
Line 1: double grade equals .895, semicolon.
Line 2: Decimal Format percent equals.
Line 3, indented once: new Decimal Format, left parentheses, double quotes, hash, dot, 0, percent, double quotes, right parentheses, semicolon.
Line 4: blank.
Line 5: System dot out dot print l n, left parentheses, double quotes, Your grade is, blank space, double quotes, plus, grade, right parentheses, semicolon.

Back to Code

The expected output is as follows. Your grade is 89.5 percent.

Back to Code

The actual output is as follows. Your grade is 0.895.

Back to Code

Line 1: area equals pi asterisk r to the power of 2. Line 2: circumference equals 2 asterisk pi asterisk r.

Back to Equation

Line 1: x 1 equals left parentheses, minus sign, b, plus, square root, left parentheses, b squared minus 4 a c, right parentheses, right parentheses, forward slash, left parentheses, a, right parentheses. Line 2: x 2 equals left parentheses, minus sign, b, minus square root, left parentheses, b squared minus 4 a c, right parentheses, right parentheses, forward slash, left parentheses, a, right parentheses.

Back to Equation

Line 1: a equals 1.0, comma, b equals 3.0, comma, c equals 2.0. Line 2: a equals 0.5, comma, b equals 0.5, comma, c equals 0.125. Line 3: a equals 1.0, comma, b equals 3.0, comma, c equals 10.0.

Back to Equation

Program code. The code contains the following keywords: import, public, class, static, String, int, return. Line 1: Forward slash, asterisk, J I Graphics Utility. Line 2, indented once: Utility class for building Java F X graphics applications. Line 3, indented once: Anderson, comma, Franceschi. Line 4: asterisk, forward slash. Line 5: blank. Line 6: import java f x dot scene dot Group, semicolon. Line 7: import java f x dot scene dot canvas dot Canvas, semicolon. Line 8: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 9: import java f x dot scene dot Scene, semicolon. Line 10: import java f x dot stage dot Stage, semicolon. Line 11: blank. Line 12:

public class J I Graphics Utility. Line 13: left brace. Line 14, indented once: public static Graphics Context set Up Graphics, left parentheses, Stage stage, comma. Line 15, indented multiple times: String title, comma, int height, comma, int width, right parentheses. Line 16, indented once: left brace. Line 17, indented twice: stage dot set Title, left parentheses, title, right parentheses, semicolon. Line 18, indented twice: Canvas canvas, equals, new Canvas, left parentheses, height, comma, width, right parentheses, semicolon. Line 19, indented twice: Graphics Context g c, equals, canvas dot get Graphics Context 2 D, left parentheses, right parentheses, semicolon. Line 20, indented twice: Group root, equals, new Group, left parentheses, canvas, right parentheses, semicolon. Line 21, indented twice: stage dot set Scene, left parentheses, new Scene, left parentheses, root, right parentheses, right parentheses, semicolon. Line 22, indented twice: stage dot show, left parentheses, right parentheses, semicolon. Line 23, indented twice: return g c, semicolon. Line 24, indented once: right brace. Line 25: right brace.

Back to Example

The table has 2 columns labelled Return value and Method name and argument list. The row entry is as follows.
Row 1: Return value: Graphics Context. Method name and argument list: set Up Graphics, left parentheses, Stage stage, comma, String title, comma, int width,

comma, int height, right parentheses, initializes a scene on the stage, sets the title of the window to title, and adds a canvas with the width and height specified, semicolon, returns a Graphics Context object.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, at symbol override, void, static, String. Line 1: Forward slash, asterisk, A Shell for creating graphics applications. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: import java f x dot application dot Application, semicolon. Line 6: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 7: import java f x dot stage dot Stage, semicolon. Line 8: blank. Line 9: public class Shell Graphics Application extends Application. Line 10: left brace. Line 11, indented once: at symbol, override. Line 12, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 13, indented once: left brace. Line 14, indented twice: 2 forward slashes, set up window title and size. Line 15, indented twice: Graphics Context g c, equals, J I Graphics Utility dot set Up Graphics, left parentheses. Line 16, indented multiple times: stage, comma, double quotes, Shell Graphics Application, double quotes, comma, 700, comma, 400, right parentheses, semicolon. Line 17: blank. Line 18, indented twice: 2 forward slashes, your drawing code goes here. Line 19, indented once: right brace. Line 20: blank. Line

21, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 22, indented once: left brace. Line 23, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 24, indented once: right brace. Line 25: right brace.

Back to Example

java c, 2 hyphens, module, hyphen, path, blank space, left angled bracket, path of Java F X lib folder, right angled bracket, 2 hyphens, add, hyphen, modules, java f x dot controls, Application Class Name dot java.

Back to Code

java c, 2 hyphens, module, hyphen, path, blank space, double quotes, uppercase c, colon, back slash, Program Files, back slash, Java F X, back slash, java f x, hyphen, s d k, hyphen, 17 point 0 point 1, back slash, lib, double quotes, 2 hyphens, add, hyphen, modules, java f x dot controls, Application Class Name dot java.

Back to Code

java c, 2 hyphens, module, hyphen, path, double quotes, uppercase c, colon, back slash, Program Files, back slash, Java F X, back slash, java f x, hyphen, s d k, hyphen, 17 point 0 point 1, back slash, lib, double quotes, 2 hyphens, add, hyphen, modules, java f x dot controls Shell Graphics Application dot java.

Back to Code

java, 2 hyphens, module, hyphen, path, left angled bracket, path of Java F X lib folder, right angled bracket, 2 hyhpens, add, hyphen, modules, java f x dot controls, Application Class Name.

Back to Code

java, 2 hyphens, module, hyphen, path, double quotes, uppercase c, colon, back slash, Program Files, back slash, Java F X, back slash, java f x, hyphen, s d k, hyphen, 17 point 0 point 1, back slash, lib, double quotes, 2 hyphens, add, hyphen, modules, java f x dot controls, Application Class Name.

Back to Code

java, 2 hyphens, module, hyphen, path, double quotes, uppercase c, colon, back slash, Program Files, back slash, Java F X, back slash, java f x, hyphen, s d k, hyphen, 17 point 0 point 1, back slash, lib, double quotes, 2 hyphens, add, hyphen, modules, java f x dot controls, Shell Graphics Application.

Back to Code

The table has 2 columns labeled Return Value and Method name and argument list. The row entries are as follows. Row 1. Return Value: void. Method name and argument list: set Stroke, left parentheses, Paint color, right parentheses, sets the current outline color to color. Row 2. Return value: void. Method name and argument list: set Fill, left parentheses, Paint color, right parentheses, sets the current fill color to color.

Back to Table

The table has 4 columns labeled Color Constant, Red, Green, and Blue. The row entries are as follows. Row 1. Color Constant: a black square is shown and the color constant is Color dot BLACK. Red: 0 0. Green: 0 0. Blue: 0 0. Row 2. Color Constant: a blue square is shown and the color constant is Color dot BLUE. Red: 0 0. Green: 0 0. Blue: F F. Row 3. Color Constant: a cyan square is shown and the color constant is Color dot CYAN. Red: 0 0. Green: F F. Blue: F F. Row 4. Color Constant: a dark gray square is shown and the color constant is Color dot DARK GRAY. Red: A 9. Green: A 9. Blue: A 9. Row 5. Color Constant: a gray square is shown and the color constant is Color dot GRAY. Red: 8 0. Green: 8 0. Blue: 8 0. Row 6. Color Constant: a green square is shown and the color constant is Color dot GREEN. Red: 0 0. Green: F F. Blue: 0 0. Row 7. Color Constant: a light gray square is shown and the color constant is color dot LIGHT GRAY. Red: D 3. Green: D 3. Blue: D 3. Row 8. Color Constant: a magenta square is shown and the color constant is Color dot MAGENTA. Red: F F. Green: 0 0. Blue: F F. Row 9. Color Constant: an orange square is shown and the color constant is Color dot ORANGE. Red: F F. Green: A 5. Blue: 0 0. Row 10. Color Constant: a pink square is shown and the color constant is Color dot PINK. Red: F F. Green: C 0. Blue: C B. Row 11. Color Constant: a red square is shown and the color constant is Color dot RED. Red: F F. Green: 0 0. Blue: 0 0. Row 12. Color Constant: a white square is shown and the color constant is Color dot WHITE. Red: F F. Green: F F. Blue: F F. Row 13. Color Constant: a yellow square is shown and the color constant is Color dot YELLOW. Red: F F. Green: F F. Blue: 0 0.

Back to Table

The table has 2 columns labeled Return Value and Method name and argument list. The row entry is as follows. Row 1. Return value: Color. Method name and argument list: r g b, left parentheses, int r r, comma, int g g, comma, int b b, right parentheses, creates a color object with an r r red component, g g green component and b b blue component.

Back to Table

The table has 2 columns labelled Return Value and Method name and argument list. The row entries are as follows.
Row 1. Return Value: void. Method name and argument list: stroke Line, left parentheses, double x Start, comma, double y Start, comma, double x End, comma, double y End, right parentheses, draws a line starting at, left parentheses, x Start, comma, y Start, right parentheses,

and ending at left parentheses, x End, comma, y End, right parentheses.
Row 2. Return Value: void. Method name and argument list: stroke R e c t, left parentheses, double x, comma, double y, comma, double width, comma, double height, right parentheses, draws the outline of a rectangle with its top left corner at left parentheses, x, comma, y, right parentheses, with the specified width and height in pixels.
Row 3. Return Value: void. Method name and argument list: fill R e c t, left parentheses, double x, comma, double y, comma, double width, comma, double height, right parentheses, draws a solid rectangle with its top left corner at, left parentheses, x, comma, y, right parentheses, with the specified width and height in pixels.
Row 4. Return Value: void. Method name and argument list: clear R e c t, left parentheses, double x, comma, double y, comma, double width, comma, double height, right parentheses, draws a solid rectangle in the current background color with its top left corner at left parentheses, x, comma, y, right parentheses, with the specified width and height in pixels.
Row 5. Return Value: void. Method name and argument list: stroke Oval, left parentheses, double x, comma, double y, comma, double width, comma, double height, right parentheses, draws the outline of an oval inside an invisible bounding rectangle with the specified width and height in pixels, semicolon, the top left corner of the bounding rectangle is, left parentheses, x, comma, y, right parentheses.

Row 6. Return Value: void. Method name and argument list: fill Oval, left parentheses, double x, comma, double y, comma, double width, comma, double height, right parentheses, draws a solid oval inside an invisible bounding rectangle with the specified width and height in pixels, semicolon, the top left corner of the bounding rectangle is left parentheses, x, comma, y, right parentheses.

Row 7. Return Value: void. Method name and argument list: stroke Text, left parentheses, String s, comma, double x, comma, double y, right parentheses, displays the string s with the left parentheses, x, comma, y, right parentheses, coordinate indicating the lower left corner of the first character, semicolon, the text is drawn in the current stroke color and line width.

Row 8. Return Value: void. Method name and argument list: fill Text, left parentheses, String s, comma, double x, comma, double y, right parentheses, displays the String s with the left parentheses, x, comma, y, right parentheses, coordinate indicating the lower left corner of the first character, semicolon, the text is drawn in the current fill color.

Row 9. Return Value: void. Method name and argument list: set Line Width, left parentheses, double width, right parentheses, sets the current stroke line width to width pixels.

Row 10. Return Value: void. Method name and argument list: set Font, left parentheses, Font f, right parentheses, sets the font to f for displaying text.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, at symbol override, void, final, int. Line 1: Forward slash, asterisk, Drawing Lines. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: import java f x dot application dot Application, semicolon. Line 6: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 7: import java f x dot scene dot paint dot Color, semicolon. Line 8: import java f x dot stage dot Stage, semicolon. Line 9: blank. Line 10: import java dot util dot Random, semicolon. Line 11: blank. Line 12: public class Drawing Lines extends Application. Line 13: left brace. Line 14, indented once: at symbol, Override. Line 15, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 16, indented once: left brace. Line 17, indented twice: final int WIDTH, equals, 700, semicolon. Line 18, indented twice: final int HEIGHT, equals, 400, semicolon. Line 19, indented twice: 2 forward slashes, set up window title and size. Line 20, indented twice: Graphics Context g c, equals, J I Graphics Utility dot set Up Graphics, left parentheses. Line 21, indented multiple times: stage, comma, double quotes, Drawing Lines, double quotes, comma, WIDTH, comma, HEIGHT, right parentheses, semicolon. Line 22: blank. Line 23, indented twice: 2 forward slashes, draw a vertical black line. Line 24, indented twice: g c dot stroke Line, left parentheses, 100, comma, 125, comma, 100, comma, 300, right parentheses, semicolon. Line 25: blank. Line 26, indented twice: 2 forward slashes, draw a horizontal red line. Line 27, indented twice: g c dot set Stroke, left parentheses, Color dot RED, right parentheses, semicolon. Line 28, indented twice: g c dot set Line Width, left parentheses, 5, right parentheses, semicolon. 2 forward slashes, widen the line. Line 29, indented twice: g c dot stroke Line, left parentheses, 300, comma, 75, comma, 550, comma, 75, right parentheses, semicolon. Line 30: blank. Line 31, indented twice: 2 forward slashes, draw a diagonal line in a random color. Line 32, indented twice: Random rand, equals, new Random, left parentheses, right parentheses, semicolon. Line 33, indented twice: int red, equals, rand dot next Int, left parentheses, 0 x F F, right parentheses, semicolon. Line 34, indented twice: int green, equals, rand dot next Int, left parentheses, 0 x F F, right parentheses, semicolon. Line 35, indented twice: int blue, equals, rand dot next Int, left parentheses, 0 x F F, right parentheses, semicolon. Line 36, indented twice: Color random Color, equals, Color dot r g b, left parentheses, red, comma, green, comma, blue, right parentheses, semicolon. Line 37, indented twice: g c dot set Stroke, left parentheses, random Color, right parentheses, semicolon. Line 38: blank. Line 39, indented twice: g c dot stroke Line, left parentheses, 0, comma, 0, comma, WIDTH minus 1, comma, HEIGHT minus 1, right parentheses, semicolon. Line 40, indented once: right brace. Line 41: blank. Line 42, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 43, indented once: left brace. Line 44, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 45, indented twice: right brace. Line 46: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, at symbol Override, void, static, String. Line 1: Forward slash, asterisk, Drawing Shapes. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: import java f x dot application dot Application, semicolon. Line 6: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 7: import java f x dot scene dot paint dot Color, semicolon. Line 8: import java f x dot stage dot Stage, semicolon. Line 9: blank. Line 10: public class Drawing Shapes extends Application. Line 11: left brace. Line 12, indented once: at symbol, Override. Line 13, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 14, indented once: left brace. Line 15, indented twice: 2 forward slashes, set up window title and size. Line 16, indented twice: Graphics Context g c, equals sign, J I Graphics Utility dot set Up Graphics, left parentheses. Line 17, indented multiple times: stage, comma, double quotes, Drawing Shapes, double quotes, comma, 700, comma, 400, right parentheses, semicolon. Line 18, indented twice: 2 forward slashes, specify colors.

Line 19, indented twice: g c dot set Fill, left parentheses, Color dot CYAN, right parentheses, semicolon. Line 20, indented twice: g c dot set Stroke, left parentheses, Color dot ORANGE, right parentheses, semicolon. Line 21: blank. Line 22, indented twice: 2 forward slashes, draw rectangles. Line 23, indented twice: g c dot stroke Rect, left parentheses, 100, comma, 50, comma, 80, comma, 200, right parentheses, semicolon. 2 forward slashes. outlined rectangle. Line 24, indented twice: g c dot fill Rect, left parentheses, 275, comma, 75, comma, 160, comma, 160, right parentheses, semicolon. 2 forward slashes, solid square. Line 25: blank. Line 26, indented twice: 2 forward slashes, draw ovals. Line 27, indented twice: g c dot fill Oval, left parentheses, 100, comma, 50, comma, 80, comma, 200, right parentheses, semicolon. 2 forward slashes, oval inside rectangle. Line 28, indented twice: g c dot stroke Oval, left parentheses, 100, comma, 275, comma, 200, comma, 80, right parentheses, semicolon. 2 forward slashes, outlined oval. Line 29: blank. Line 30, indented twice: 2 forward slashes, draw circle using center point and radius. Line 31, indented twice: g c dot set Fill, left parentheses, Color dot MEDIUM SEA GREEN, right parentheses, semicolon. Line 32, indented twice: int center X, equals, 550, comma, center Y, equals, 275, semicolon. Line 33, indented once: int radius, equals, 75, semicolon. Line 34, indented once: g c dot fill Oval, left parentheses, center X minus radius, comma, center Y minus radius, comma. Line 35, indented multiple times: radius asterisk 2, comma, radius asterisk

2, right parentheses, semicolon. Line 36, indented once: right brace. Line 37: blank. Line 38, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 39, indented once: left brace. Line 40, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 41, indented once: right brace. Line 42: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, at symbol override, void, static, String. Line 1: Forward slash, asterisk, Drawing Text. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: import java f x dot application dot Application, semicolon. Line 6: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 7: import java f x dot scene dot paint dot Color, semicolon. Line 8: import java f x dot scene dot text dot Font, semicolon. Line 9: import java f x dot stage dot Stage, semicolon. Line 10: blank. Line 11: public class Drawing Text extends Application. Line 12: left brace. Line 13, indented once: at symbol, Override. Line 14, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 15, indented once: left brace. Line 16, indented twice: 2 forward slashes, set up window title and size. Line 17, indented twice: Graphics Context g c, equals, J I Graphics Utility dot set Up Graphics, left parentheses.

Line 18, indented multiple times: stage, comma, double quotes, Drawing Text, double quotes, comma, 700, comma, 400, right parentheses, semicolon. Line 19: blank. Line 20, indented twice: String message, equals, double quotes, Programming is not a spectator sport, exclamation mark, double quotes, semicolon. Line 21, indented twice: g c dot set Fill, left parentheses, Color dot RED, right parentheses, semicolon. Line 22: blank. Line 23, indented twice: g c dot fill Text, left parentheses, message, comma, 100, comma, 100, right parentheses, semicolon. Line 24: blank. Line 25, indented twice: Font large Font, equals, new Font, left parentheses, 28, right parentheses, semicolon. Line 26, indented twice: g c dot set Font, left parentheses, large Font, right parentheses, semicolon. Line 27, indented twice: g c dot set Stroke, left parentheses, Color dot RED, right parentheses, semicolon. Line 28: blank. Line 29, indented twice: g c dot stroke Text, left parentheses, message, comma, 100, comma, 250, right parentheses, semicolon. Line 30, indented once: right parentheses. Line 31: blank. Line 32, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 33, indented once: left brace. Line 34, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 35, indented once: right brace. Line 36: right brace.

Back to Example

The constructor is as follows: Font, left parentheses, double size, right parentheses, creates a Font object with a height of size in the default typeface.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, at symbol Override, void, int, static. Line 1: Forward slash, asterisk, Drawing a Sprite. Line 3, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: import java f x dot application dot Application, semicolon. Line 6: import java f x dot scene dot canvas dot Canvas, semicolon. Line 7: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 8: import java f x dot scene dot paint dot Color, semicolon. Line 9: import java f x dot stage dot Stage, semicolon. Line 10: blank. Line 11: public class Drawing A Sprite extends Application. Line 12: left brace. Line 13, indented once: at symbol, Override. Line 14, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 15, indented once: right parentheses. Line 16, indented twice: 2 forward slashes, set up window title and size. Line 17, indented twice: Graphics Context g c, equals, J I Graphics Utility dot set Up Graphics, left parentheses. Line 18, indented multiple times: stage, comma, double quotes, Drawing a Sprite, double quotes, comma, 700, comma, 400, right parentheses, semicolon. Line 19: blank. Line 20, indented twice: 2 forward slashes, drawing the sprite. Line 21, indented twice: int s X, equals, 275, semicolon. Line 22, indented twice: int s Y, equals, 100, semicolon. Line 23, indented twice: g c dot set Fill, left parentheses, Color dot CORAL, right parentheses, semicolon. 2 forward slashes, body. Line 24, indented twice: g c dot fill Oval, left parentheses, s X, comma, s Y plus 20, comma, 120, comma, 160, right parentheses, semicolon. Line 25, indented twice: g c dot set Fill, left parentheses, Color dot DARK GOLDEN ROD, right parentheses, semicolon. 2 forward slashes, hat. Line 26, indented twice: g c dot fill Rect, left parentheses s X plus 30, comma, s Y, comma, 60, comma, 30, right parentheses, semicolon. Line 27, indented twice: g c dot set Stroke, left parentheses, Color dot DARK GOLDEN ROD, right parentheses, semicolon. 2 forward slashes, hat brim. Line 28, indented twice: g c dot set Line Width, left parentheses, 3, right parentheses, semicolon. Line 29, indented twice: g c dot stroke Line, left parentheses, s X, comma, s Y plus 30, comma, s X plus 120, comma, s Y plus 30, right parentheses, semicolon. Line 30, indented twice: g c dot set Fill, left parentheses, Color dot CHOCOLATE, right parentheses, semicolon. 2 forward slashes, eye. Line 31, indented twice: g c dot fill Oval, left parentheses, s X plus 80, comma, s Y plus 60, comma, 24, comma, 16, right parentheses, semicolon. Line 32, indented twice: g c dot set Fill, left parentheses, Color dot DARK SALMON, right parentheses, semicolon. 2 forward slashes, feet. Line 33, indented twice: g c dot set Line Width, left parentheses, 1, right parentheses, semicolon. Line 34, indented twice: g c dot fill Oval, left parentheses, s X plus 64, comma, s Y plus 166, comma, 60, comma, 16, right parentheses, semicolon. Line 35, indented twice: g c dot stroke Oval, left parentheses, s X plus 64, comma, s Y plus 166, comma, 60, comma, 16, right parentheses, semicolon. Line 36, indented twice: g c dot fill Oval, left parentheses, s X plus 36, comma, s Y plus 170, comma, 60, comma, 16, right parentheses, semicolon. Line 37, indented twice: g c dot stroke Oval, left parentheses, s X plus 36, comma, s Y plus 170, comma, 60, comma, 16, right parentheses, semicolon. Line 38, indented once: right brace. Line 39: blank. Line 40, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 41, indented once: left brace. Line 42, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 43, indented once: right brace. Line 44: right brace.

Back to Example

The table has 2 columns labeled Return Value and Method Name and Argument List. The row entries are as follows. Row 1. Return Value: void. Method Name and Argument List: begin Path, left parentheses, right parentheses, sets the drawing path to empty. Row 2. Return Value: void. Method Name and Argument List: move To, left parentheses, double x, comma, double y, right parentheses, moves to the, left parentheses, x, comma, y, right parentheses, location without adding a line to the path. Row 3. Return Value: void. Method Name and Argument List: line To, left parentheses, double x, comma, double y, right parentheses, adds a line to the current path from the previous location to, left parentheses, x, comma, y, right parentheses. Row 4. Return Value: void. Method Name and Argument List: close Path, left parentheses, right parentheses, closes the path by adding a line from the previous location to the first location of the path. Row 5. Return Value: void. Method Name and Argument List: stroke, left parentheses, right parentheses, draws the path using the current stroke color. Row 6. Return Value: void. Method Name and Argument List: fill, left parentheses, right parentheses, draws the path using the current fill color.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, at symbol Override, void, int, static. Line 1: Forward slash, asterisk, Drawing Shapes Using Paths. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: import java f x dot application dot Application, semicolon. Line 6: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 7: import java f x dot scene dot paint dot Color, semicolon. Line 8: import java f x dot stage dot Stage, semicolon. Line 9: blank. Line 10: public class Drawing Paths extends Application. Line 11: left brace. Line 12, indented once: at symbol, Override. Line 13, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 14, indented once: left brace. Line 15, indented twice: 2 forward slashes, set up window title and size. Line 16, indented twice: Graphics Context g c, equals, J I Graphics Utility dot set Up Graphics, left parentheses. Line 17, indented multiple times: stage, comma, double quotes, Using Paths, double quotes, comma, 700, comma, 400, right parentheses, semicolon. Line 18: blank. Line 19, indented twice: 2 forward slashes, drawing a triangle. Line 20, indented twice: 2 forward slashes, 1, period, create the path. Line 21, indented twice: int start X, equals, 175, semicolon. Line 22, indented twice: int start Y, equals, 120, semicolon. Line 23, indented twice: g c dot begin Path, left parentheses, right parentheses, semicolon. Line 24, indented twice: g c dot move To, left parentheses, start X, comma, start Y, right parentheses, semicolon. Line 25, indented twice: g c dot line To, left parentheses, start X minus 50, comma, start Y plus 175, right parentheses, semicolon. Line 26, indented twice: g c dot line To, left parentheses, start X plus 50, comma, start Y plus 175, right parentheses, semicolon. Line 27, indented twice: g c dot close Path, left parentheses, right parentheses, semicolon. Line 28: blank. Line 29, indented twice: 2 forward slashes, 2 draw the path. Line 30, indented twice: g c dot set Stroke, left parentheses, Color dot CRIMSON, right parentheses, semicolon. Line 31, indented twice: g c dot stroke, left parentheses, right parentheses, semicolon. Line 32: blank. Line 33, indented twice: 2 forward slashes. draw a hexagon. Line 34, indented twice: 2 forward slashes. 1, period, create the path. Line 35, indented twice: start X, equals, 350, semicolon. Line 36, indented twice: start Y, equals, 200, semicolon. Line 37, indented twice: g c dot begin Path, left parentheses, right parentheses, semicolon. Line 38, indented twice: g c dot move To, left parentheses, start X, comma, start Y, right parentheses, semicolon. Line 39, indented twice: g c dot line To, left parentheses, start X plus 50, comma, start Y minus 100, right parentheses, semicolon. Line 40, indented twice: g c dot line To, left parentheses, start X plus 175, comma, start Y minus 100, right parentheses, semicolon. Line 41, indented twice: g c dot line To, left parentheses, start X, plus, 225, comma, start Y, right parentheses, semicolon. Line 42, indented twice: g c dot line To, left parentheses, start X plus 175, comma, start Y plus 100, right parentheses, semicolon. Line 43, indented twice: g c dot line To, left parentheses, start X plus 50, comma, start Y plus 100, right parentheses, semicolon. Line 44, indented twice: g c dot close Path, left parentheses, right parentheses, semicolon. Line 45: blank. Line 46, indented twice: 2 forward slashes. 2, period, draw the path. Line 47, indented twice: g c dot set Fill, left parentheses, Color dot CORN FLOWER BLUE, right parentheses, semicolon. Line 48, indented twice: g c dot fill, left parentheses, right parentheses, semicolon. Line 49, indented once: right brace. Line 50: blank. Line 51, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 52, indented once: left brace. Line 53, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon.

Line 54, indented once: right brace. Line 55: right brace.

Back to Example

g c dot stroke Line, left parentheses, 100, comma, 200, comma, 300, comma, 200, right parentheses, semicolon.

Back to Code

g c dot stroke R e c t, left parentheses, 10, comma, 20, comma, 100, comma, 200, right parentheses, semicolon.

Back to Code

g c dot stroke R e c t, left parentheses, 10, comma, 20, comma, 100, comma, 200, right parentheses, semicolon.

Back to Code

Line 1: g c dot set Fill, left parentheses, Color dot BLUE, right parentheses, semicolon. Line 2: g c dot fill R e c t, left parentheses, 10, comma, 20, comma, 100, comma, 200, right parentheses, semicolon.

Back to Code

g c dot stroke Line, left parentheses, 50, comma, 20, comma, 50, comma, 350, right parentheses, semicolon.

Back to Code

g c dot fill R e c t, left parentheses, 10, comma, 20, comma, 250, comma, 350, right parentheses, semicolon.

Back to Code

g c dot fill R e c t, left parentheses, 10, comma, 20, comma, 250, comma, 350, right parentheses, semicolon.

Back to Code

g c dot stroke R e c t, left parentheses, 10, comma, 20, comma, 250, comma, 350, right parentheses, semicolon.

Back to Code

Line 1: 2 forward slashes. assume you have a Graphics Context object named g c. Line 2: 2 forward slashes. your code goes here.

Back to Code

Line 1: 2 forward slashes. assume you have a Graphics Context object named g c. Line 2: 2 forward slashes. your code goes here.

Back to Code

Line 1: 2 forward slashes. assume you have a Graphics Context object named g c. Line 2: 2 forward slashes. your code goes here.

Back to Code

Line 1: 2 forward slashes. assume you have a Graphics Context object named g c. Line 2: 2 forward slashes. your code goes here.

Back to Code

Line 1: 2 forward slashes. assume you have a Graphics Context object named g c. Line 2: 2 forward slashes. your code goes here.

Back to Code

Line 1: g c dot set Stroke, left parentheses, Color dot BLUE, right parentheses, semicolon. Line 2: g c dot fill R e c t, left parentheses, 10, comma, 20, comma, 100, comma, 200, right parentheses, semicolon.

Back to Code

g c dot fill Text, left parentheses, single quote, Find the bug, single quote, comma, 100, comma, 200, right parentheses, semicolon.

Back to Code

Program code. The words in the variable names are merged. The code contains the following keywords: import, public, class, extends, at Override, void, static.
Line 1: import java f x dot application dot Application, semicolon.
Line 2: import java f x dot scene dot canvas dot Graphics Context, semicolon.
Line 3: import java f x dot stage dot Stage, semicolon.
Line 4: public class My Drawing App extends Application.
Line 5: left brace.

Line 6, indented once: at Override.
Line 7, indented once: public void start, left parentheses, Stage stage, right parentheses.
Line 8, indented once: left brace.
Line 9, indented twice: Graphics Context g c equals J I Graphics Utility dot set Up Graphics, left parentheses.
Line 10, indented multiple times: stage, comma, double quotes, Tester, double quotes, comma, 700, comma, 400, right parentheses, semicolon.
Line 11, indented twice: g c dot Stroke R e c t, left parentheses, 725, comma, 200, comma, 50, comma, 50, right parentheses, semicolon.
Line 12, indented once: right brace.
Line 13, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses.
Line 14, indented once: left brace.
Line 15, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon.
Line 16, indented once: right brace.
Line 17: right brace.

Back to Code

The statement is as follows. Color c equals Color dot r g b, left parentheses, 1.4, comma, 234, comma, 23, right parentheses, semicolon. 2 forward slashes. line 10.

Back to Code

The compiler message is as follows.
Line 1: My Application dot java, colon, 10, colon, error, colon, incompatible types, colon, possible.
Line 2: lossy conversion from double to int.
Line 3: Color c equals Color dot r g b, left parentheses, 1.4, comma, 234, comma, 23, right parentheses, semicolon. 2 forward slashes. line 10.
Line 4: A caret sign under 1.4 in Line 3.
Line 5: 1 error.

Back to Code

The statement is as follows. Color c equals Color dot Blue, semicolon. 2 forward slashes. line 10.

Back to Code

The compiler message is as follows.
Line 1: My Application dot java, colon, 10, colon, error, colon, cannot find symbol.
Line 2, indented once: Color c equals Color dot Blue, semicolon. 2 forward slashes. line 10.
Line 3: a caret sign under the dot in Line 2.
Line 4: symbol, colon, variable Blue.
Line 5: location, colon, class Color.
Line 6: 1 error.

Back to Code

The table has 3 columns labeled Equality Operator, Type, and Meaning. The row entries are as follows. Row 1.

Equality Operator: 2 equals sign. Type: binary. Meaning: is equal to. Row 2. Equality Operator: exclamation mark, equals sign. Type: binary. Meaning: is not equal to.

Back to Table

The table has 3 columns labeled Relational Operator, Type, and Meaning. The row entries are as follows. Row 1. Relational Operator: less than sign. Type: binary. Meaning: is less than. Row 2. Relational Operator: less than or equal to. Type: binary. Meaning: is less than or equal to. Row 3. Relational Operator: greater than sign. Type: binary. Meaning: is greater than. Row 4. Relational Operator: greater than or equal to. Type: binary. Meaning: is greater than or equal to.

Back to Table

The expression, left parentheses, age less than 32, right parentheses, will evaluate to false. The expression, left parentheses, age less than or equal to 32, right parentheses, will evaluate to true. The expression, left parentheses, age greater than 32, right parentheses, will evaluate to false, and. The expression, left parentheses, age greater than or equal to 32, right parentheses, will evaluate to true.

Back to Code

The table has 3 columns labeled Logical Operator, Type, and Meaning. The row entries are as follows.

Row 1. Logical Operator: exclamation mark. Type: unary. Meaning: NOT.
Row 2. Logical Operator: 2 ampersand signs. Type: binary. Meaning: AND.
Row 3. Logical Operator: 2 pipes. Type: binary. Meaning: OR.

Back to Table

The table has 2 columns labeled Operands and Operations. The column labeled Operands is subdivided into 2 columns which are labeled a and b. The column labeled Operations is divided into 3 columns which are labelled exclamation mark a, a 2 ampersand signs b, and a 2 pipes b. The row entries are as follows.
Row 1. a: true. b: true. exclamation mark a: false. a 2 ampersand signs b: true. a 2 pipes b: true.
Row 2. a: true. b: false. exclamation mark a: false. a 2 ampersand signs b: false. a 2 pipes b: true.
Row 3. a: false. b: true. exclamation mark a: true. a 2 ampersand signs b: false. a 2 pipes b: true.
Row 4. a: false. b: false. exclamation mark a: true. a 2 ampersand signs b: false. a 2 pipes b: false.

Back to Table

Row 1. Operator Hierarchy: left parentheses, right parentheses. Order of Same Statement Evaluation: left to right. Operation: parentheses for explicit grouping.
Row 2. Operator Hierarchy: plus plus, minus minus. Order

of Same Statement Evaluation: right to left. Operation: shortcut post increment or post decrement.
Row 3. Operator Hierarchy: plus plus, minus minus, exclamation mark. Order of Same Statement Evaluation: right to left. Operation: shortcut pre increment or pre decrement, logical unary NOT.
Row 4. Operator Hierarchy: asterisk, forward slash, percentage sign. Order of Same Statement Evaluation: left to right. Operation: multiplication, division, modulus.

Back to Table

Program code. The code contains the following keywords: public, class, static, void, int, boolean. Line 1: forward slash, asterisk, Using Logical Operators. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Logical Operators. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: int age, equals, 75, semicolon. Line 10, indented once: boolean test, semicolon. Line 11: blank. Line 12, indented once: test, equals, left parentheses, age greater than 18, ampersand, ampersand, age less than 65, right parentheses, semicolon. Line 13, indented once: System dot out dot print l n, left parentheses, age, plus, double quotes, right angled bracket, 18, ampersand, ampersand, double quotes, plus, age, plus, double quotes, left angled bracket, 65 is, double quotes, plus, test, right parentheses, semicolon. Line 14: blank. Line 15, indented once: 2 forward slashes. short circuitry with AND. Line 16, indented once: test, equals, left parentheses, age, less than, 65, ampersand, ampersand, age, greater than, 18, right parentheses, semicolon. Line 17, indented once: System dot out dot print l n, left parentheses, age, plus, double quotes, left angled bracket, 65, ampersand, ampersand, double quotes, plus, age, plus, double quotes, right angled bracket, 18 is, double quotes, plus, test, right parentheses, semicolon. Line 18: blank. Line 19, indented once: 2 forward slashes, short circuitry with OR. Line 20, indented once: test, equals, left parentheses, age, greater than, 65, 2 pipes, age less than 18, right parentheses, semicolon. Line 21, indented once: System dot out dot print l n, left parentheses, age, plus, double quotes, right angled bracket, 65, 2 pipes, double quotes, plus, age, plus, double quotes, left angled bracket, 18 is, double quotes, plus, test, right angled bracket, semicolon. Line 22: blank. Line 23, indented once: 2 forward slashes, AND has higher precedence than OR. Line 24, indented once: test, equals, left parentheses, age greater than 65, 2 pipes, age less than 18, 2 ampersands, false, right parentheses, semicolon. Line 25, indented once: System dot out dot print l n, left parentheses, age, plus, double quotes, right angled bracket, 65, 2 pipes, double quotes, plus, age. Line 26, indented multiple times: plus, double quotes, left angled bracket, 18, 2 ampersands, false is, double quotes, plus, test, right parentheses, semicolon. Line 27: blank. Line

28, indented once: 2 forward slashes, use of parentheses to force order of execution. Line 29, indented once: test, equals, left parentheses, left parentheses, age greater than 65, 2 pipes, age less than 18, right parentheses, 2 ampersands, false, right parentheses, semicolon. Line 30, indented once: System dot out dot print l n, left parentheses, double quotes, left parentheses, double quotes, plus, age, plus, double quotes, right angled bracket, 65, 2 pipes, double quotes, plus, age. Line 31, indented multiple times: plus, double quotes, left angled bracket, 18, right parentheses, 2 ampersands, false is, double quotes, plus, test, right parentheses, semicolon. Line 32, indented once: right brace. Line 33: right brace.

Back to Example

x, left angled bracket, y, 2 ampersand signs, z. 2 forward slashes. incorrect comparison of x to y and z.

Back to Code

x, left angled bracket, y, 2 ampersand signs, x, left angled bracket, z.

Back to Code

The table has 2 columns labelled Equivalent Expressions and English Meaning. The row entries are as follows. Row 1. Equivalent Expressions: left parentheses, flag 1, 2 equals sign, false, right parentheses, 2 pipes, left parentheses, flag 2, 2 equals sign, false, right

parentheses. English Meaning: flag 1 is false OR flag 2 is false.
Row 2. Equivalent Expressions: exclamation mark, flag 1, 2 pipes, exclamation mark, flag 2. English Meaning: exclamation mark flag 1 is true OR exclamation mark flag 2 is true.
Row 3. Equivalent Expressions: exclamation mark, left parentheses, flag 1, 2 ampersand signs, flag 2, right parentheses. English Meaning: not both flag 1 and flag 2 are true.

Back to Table

The first law is as follows. 1. NOT, left parentheses, A AND B, right parentheses, equals sign, left parentheses, NOT A, right parentheses, OR, left parentheses, NOT B, right parentheses. The second law is as follows. 2. NOT, left parentheses, A OR B, right parentheses, equals, left parentheses, NOT A, right parentheses, AND, left parentheses, NOT B, right parentheses.

Back to Code

exclamation mark, left parentheses, a, 2 ampersand signs, b, right parentheses, is equivalent to, exclamation mark, a, 2 pipes, exclamation mark b.

Back to Code

exclamation mark, left parentheses, a, 2 pipes, b, right parentheses, is equivalent to, exclamation mark, a, 2 ampersand signs, exclamation mark, b.

Back to Code

The table has 10 columns labelled a, b, exclamation mark a, exclamation mark b, a 2 ampersand signs b, a 2 pipes b, exclamation mark, left parentheses, a, 2 ampersand signs, b, right parentheses, exclamation mark 1, 2 pipes, exclamation mark b, exclamation mark, left parentheses, a, 2 pipes, b, right parentheses, exclamation mark a, 2 ampersand signs, exclamation mark b. The row entries are as follows.
Row 1. a: true. b: true. exclamation mark a: false. exclamation mark b: false. a 2 ampersand signs b: true. a 2 pipes b: true. exclamation mark, left parentheses, a 2 ampersand signs b, right parentheses: false. exclamation mark a 2 pipes, exclamation mark b: false. exclamation mark, left parentheses, a 2 pipes, b, right parentheses: false. exclamation mark a 2 ampersand signs exclamation mark b: false.
Row 2. a: true. b: false. exclamation mark a: false. exclamation mark b: true. a 2 ampersand signs b: false. a 2 pipes b: true. exclamation mark, left parentheses, a 2 ampersand signs b, right parentheses: true. exclamation mark a 2 pipes, exclamation mark b: true. exclamation mark, left parentheses, a 2 pipes, b, right parentheses: false. exclamation mark a 2 ampersand signs exclamation mark b: false.
Row 3. a: false. b: true. exclamation mark a: true. exclamation mark b: false. a 2 ampersand signs b: false.

a 2 pipes b: true. exclamation mark, left parentheses, a 2 ampersand signs b, right parentheses: true. exclamation mark a 2 pipes, exclamation mark b: true. exclamation mark, left parentheses, a 2 pipes, b, right parentheses: false. exclamation mark a 2 ampersand signs exclamation mark b: false.
Row 4. a: false. b: false. exclamation mark a: true. exclamation mark b: true. a 2 ampersand signs b: false. a 2 pipes b: false. exclamation mark, left parentheses, a 2 ampersand signs b, right parentheses: true. exclamation mark a 2 pipes, exclamation mark b: true. exclamation mark, left parentheses, a 2 pipes, b, right parentheses: true. exclamation mark a 2 ampersand signs exclamation mark b: true.

Back to Table

The table has 2 columns labelled Expression and exclamation mark, left parentheses, expression, right parentheses. The row entries are as follows.
Row 1. Expression: a 2 equals sign b. exclamation mark, left parentheses, Expression, right parentheses: a exclamation mark equals sign b.
Row 2. Expression: a exclamation mark equal sign b. exclamation mark, left parentheses, Expression, right parentheses: a 2 equals sign b.
Row 3. Expression: a left angled bracket b. exclamation mark, left parentheses, Expression, right parentheses: a right angled bracket equals sign b.
Row 4. Expression: a right angled bracket equals sign b.

exclamation mark, left parentheses, Expression, right parentheses: a left angled bracket b.
Row 5. Expression: a right angled bracket b. exclamation mark, left parentheses, Expression, right parentheses: a left angled bracket equals sign b.
Row 6. Expression: a left angled bracket equals sign b. exclamation mark, left parentheses, Expression, right parentheses: a right angled bracket b.

Back to Table

The table has 2 columns labelled Equivalent Expressions and English Meaning. The row entries are as follows.
Row 1. Equivalent Expressions: left parentheses, age, left angled bracket, equals sign, 18, 2 pipes, age, right angled bracket, equals sign, 65, right angled bracket. English Meaning: age is less than or equal to 18 or age is greater than or equal to 65.
Row 2. Equivalent Expressions: exclamation mark, left parentheses, age, right angled bracket, 18, 2 ampersand signs, age, left angled bracket, 65, right angled bracket. English Meaning: age is not between 18 and 65.
Row 3. Equivalent Expressions: exclamation mark, left parentheses, age, right angled bracket, 18, right parentheses, 2 pipes, exclamation mark, left parentheses, age, left angled bracket, 65, right angled bracket. English Meaning: age is not greater than 18 or age is not less than 65.

Back to Table

The pattern is as follows. Line 1: if, left parentheses, condition, right parentheses. Line 2: left brace. Line 3, indented once: true block. Line 4: right brace. Line 5: next statement.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, String, int, if. Line 1: forward slash, asterisk, Using if to calculate a final test grade. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: import java dot util dot Scanner, semicolon. Line 5: blank. Line 6: public class Test Grade. Line 7: left brace. Line 8, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: System dot out dot print l n, left parentheses, double quotes, Enter your test grade, right parentheses, double quotes, right parentheses, semicolon. Line 13, indented twice: int grade, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: System dot out dot print, left parentheses, double quotes, Enter your extra credit, right angled bracket, double quotes, right parentheses, semicolon. Line 16, indented twice: int extra Credit, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 17: blank. Line 18, indented twice: if, left parentheses, extra Credit greater than 0, right parentheses. Line 19, indented thrice: grade, plus, equals, extra Credit, semicolon. Line 20: blank. Line 21, indented twice: System dot out dot print l n, left parentheses, double quotes, Your final test grade is, double quotes. Line 22, indented multiple times: plus, grade, right parentheses, semicolon. Line 23, indented once: right brace. Line 24: right brace.

Back to Example

Program code. The code contains the following keywords: if, System, out, print l n. Line 1: if, left parentheses, grade, right angled bracket, equals sign, 60, right parentheses, semicolon. 2 forward slashes. incorrect to place semicolon here. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, You passed, double quotes, right parentheses, semicolon.

Back to Code

System dot out dot print l n, left parentheses, double quotes, You passed, double quotes, right parentheses, semicolon.

Back to Code

The patterns is as follows. Line 1: if, left parentheses, condition, right parentheses. Line 2: left brace. Line 3, indented once: true block. Line 4: right brace. Line 5: else.

Line 6: left brace. Line 7, indented once: false block. Line 8: right brace. Line 9: next statement.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, String, int, if, else. Line 1: forward slash, asterisk, Using if, forward slash, else. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Passing Grade. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10: left brace. Line 11, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 12: blank. Line 13, indented twice: System dot out dot print, left parentheses, double quotes, Enter a grade, right angled bracket, double quotes, right parentheses, semicolon. Line 14, indented twice: int grade, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 15: blank. Line 16, indented twice: String message, semicolon. Line 17, indented twice: if, left parentheses, grade greater than or equal to 60, right parentheses. Line 18, indented thrice: message, equals, double quotes, You passed, double quotes, semicolon. Line 19, indented twice: else. Line 20, indented thrice: message, equals, double quotes, You failed, double quotes, semicolon. Line 21: blank. Line 22, indented twice: System dot out dot print l n, left parentheses, message, right parentheses, semicolon. Line 23, indented once: right brace. Line 24: right brace.

Back to Example

Line 1: if, left parentheses, grade greater than or equal to 60, right parentheses. Line 2, indented once: message, equals, double quotes, You passed, double quotes, semcolon. Line 3: blank. Line 4: if, left parentheses, grade less than 60, right parentheses. Line 5, indented once: message, equals, double quotes, You failed, double quotes, semicolon.

Back to Code

Program code. The code contains the following keywords: if, String, else, System, out, print l n. Line 1: if, left parentheses, grade, right angled bracket, equals sign, 60, right parentheses. Line 2: left brace. Line 3, indented once: String message equals, double quotes, You passed, double quotes, semicolon. Line 4: right brace. Line 5: else. Line 6, indented once: message equals double quotes, You failed, double quotes, semicolon. Line 7: blank. Line 8: System dot out dot print l n, left parentheses, message, right parentheses, semicolon.

Back to Code

The pattern is as follows. Line 1: if, left parentheses, condition 1, right parentheses. Line 2: left brace. Line 3, indented once: true block for condition 1. Line 4: right brace. Line 5: else if, left parentheses, condition 2, right parentheses. Line 6: left brace. Line 7, indented once: true block for condition 2. Line 8: right brace. Line 9: dot, dot, dot. Line 10: else if, left parentheses, condition n, right parentheses. Line 11: left brace. Line 12, indented once: true block for condition n. Line 13: right brace. Line 14: else. Line 15: left brace. Line 16, indented once: false block for all conditions being false. Line 17: right brace. Line 18: next statement.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, String, char, int, if, else. Line 1: forward slash, asterisk. A program to translate a numeric grade into a letter grade. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Letter Grade. Line 8: left brace. Line 9, indented once: pubic static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 12: blank. Line 13, indented twice: char letter Grade, semicolon. Line 14: blank. Line 15, indented twice: System dot out dot print, left parentheses, double quotes, Enter your test grade, colon, double quotes, right parentheses, semicolon Line 16, indented twice: int grade, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 17: blank. Line 18, indented twice: if, left parentheses, grade, greater than or equal to 90, right parentheses. Line 19, indented thrice: letter Grade, equals, single quote, uppercase a, single quote, semicolon. Line 20: blank. Line 21, indented twice: else if, left parentheses, grade greater than or equal to 80, right parentheses. Line 22, indented thrice: letter Grade, equals, single quote, uppercase b, single quote, semicolon. Line 23: blank. Line 24, indented twice: else if, left parentheses, grade greater than or equal to 70, right parentheses. Line 25, indented thrice: letter Grade, equals, single quote, uppercase c, single quote, semicolon. Line 26: blank. Line 27, indented twice: else if, left parentheses, grade greater than or equal to 60, right parentheses. Line 28, indented thrice: letter Grade, equals, single quote, uppercase d, single quote, semicolon. Line 29: blank. Line 30, indented twice: else, 2 forward slashes, grade fits none of the conditions. Line 31, indented thrice: letter Grade, equals, single quote, uppercase f, single quote, semicolon. Line 32: blank. Line 33, indented twice: System dot out dot print l n, left parentheses, double quotes, Your test grade of, double quotes, plus, grade. Line 34, indented multiple times: plus, double quotes, is a letter grade of, double quotes, plus, letter Grade, right parentheses, semicolon. Line 35, indented once: right brace. Line 36: right brace.

Back to Example

if, left parentheses, grade, right angled bracket, equals sign, 80, right parentheses.

Back to Code

if, left parentheses, grade, right angled bracket, equals sign, 80, 2 ampersand signs, grade, left angled bracket, 90, right parentheses.

Back to Code

Line 1: read number 1. Line 2: read number 2. Line 3: read number 3. Line 4: blank. Line 5: if number 1 is less than number 2. Line 6, indented once: smallest is number 1. Line 7: else. Line 8, indented once: smallest is number 2. Line 9: if number 3 is less than smallest. Line 10, indented once: smallest is number 3.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, int, new, if, else. Line 1: forward slash, asterisk, Find the smallest of three integers. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Find Smallest. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: int smallest, semicolon. Line 12, indented twice: int num 1, comma, num 2, comma, num 3, semicolon. Line 13: blank. Line 14, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 15: blank. Line 16, indented twice: System dot out dot print, left parentheses, double quotes, Enter the first integer, colon, double quotes, right parentheses, semicolon. Line 17, indented twice: num 1, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 18, indented twice: System dot out dot print, left parentheses, double quotes, Enter the second integer, colon, double quotes, right parentheses, semicolon. Line 19, indented twice: num 2, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 20, indented twice: System dot out dot print, left parentheses, double quotes, Enter the third integer, colon, double quotes, right parentheses, semicolon. Line 21, indented twice: num 3, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 22: blank. Line 23, indented twice: if, left parentheses, num 1 less than num 2, right parentheses. Line 24, indented thrice: smallest, equals, num 1, semicolon. Line 25, indented twice: else. Line 26, indented thrice: smallest, equals, num 2, semicolon. Line 27: blank. Line 28, indented twice: if, left parentheses, num 3 less than smallest, right parentheses. Line 29, indented thrice: smallest, equals, num 3, semicolon. Line

30: blank. Line 31, indented twice: System dot out dot print l n, left parentheses, double quotes, The smallest is, double quotes, plus, smallest, right parentheses, semicolon. Line 32, indented once: right brace. Line 33: right brace.

Back to Example

Program code. The code contains the following keywords: if, else, System, out, print l n. Line 1: if, left parentheses, x, 2 equals sign, 2, right parentheses. Line 2, indented once: if, left parentheses, y, 2 equals sign, x, right parentheses. Line 3, indented twice: System dot out dot print l n, left parentheses, double quotes, x and y equal 2, double quotes, right parentheses, semicolon. Line 4, indented once: else. Line 5, indented twice: System dot out dot print l n, left parentheses, double quotes, x equals 2, comma, but y does not, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: if, else, System, out, print l n. Line 1: left brace. Line 2, indented once: if, left parentheses, y, 2 equals sign, x, right parentheses. Line 3, indented twice: System dot out dot print l n, left parentheses, double quotes, x and y equals 2, double quotes, right parentheses, semicolon. Line 4: right brace. Line 5: else. Line 6, indented once: System dot out dot print l n, left parentheses, double quotes, x does not equal 2, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: if, else, System, out, print l n.
Line 1: if, left parentheses, x, 2 equals sign, 2, right parentheses.
Line 2, indented once: if, left parentheses, y, 2 equals sign, x, right parentheses.
Line 3, indented twice: System dot out dot print l n, left parentheses, double quotes, x and y equal 2, double quotes, right parentheses, semicolon.
Line 4: blank.
Line 5, indented once: else. 2 forward slashes, matches y, 2 equals sign, x.
Line 6, indented twice: System dot out dot print l n, left parentheses, double quotes, y does not equal 2, double quotes, right parentheses, semicolon.
Line 7: blank.
Line 8: else. 2 forward slashes, matches x, 2 equals sign, 2.
Line 9, indented once: System dot out dot print l n, left parentheses, double quotes, x does not equals 2, double quotes, right parentheses, semicolon.
Line 10: blank.
Line 11: else. 2 forward slashes, no matching if, exclamation mark.
Line 12, indented once: System dot out dot print l n, left

parentheses, double quotes, x and y are not equal, double quotes, right parentheses, semicolon.

Back to Code

Line 1: generate a secret random number between 1 and 10. Line 2: prompt the user for a guess. Line 3: if guess is not between 1 and 10. Line 4, indented once: print message. Line 5: else. Line 6, indented once: if guess equals the secret number. Line 7, indented twice: print congratulations. Line 8, indented once: else. Line 9, indented twice: print the secret number. Line 10, indented twice: if guess is not within 3 numbers. Line 11, indented thrice: print, double quotes, You missed it by a mile, exclamation mark, double quotes. Line 12, indented twice: else. Line 13, indented thrice: print, double quotes, You were close, period, double quotes. Line 14, indented twice: print, double quotes, Better luck next time, period, double quotes.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, String, int, if, else. Line 1: forward slash, asterisk, Guess a number between 1 and 10. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: import java dot util dot Random, semicolon. Line 6: import java dot util dot Scanner, semicolon. Line 7: blank. Line 8: public class Guess A Number. Line 9: left brace.

Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: Random random, equals, new Random, left parentheses, right parentheses, semicolon. Line 13, indented twice: int secret Number, equals, random dot next Int, left parentheses, 10, right parentheses, plus, 1, semicolon. Line 14: blank. Line 15, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 16: blank. Line 17, indented twice: System dot out dot print, left parentheses, double quotes, I'm thinking of a number, double quotes. Line 18, indented multiple times: plus, double quotes, between 1 and 10. What is your guess, question mark, double quotes, right parentheses, semicolon. Line 19, indented twice: int guess, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 20: blank. Line 21, indented twice: if, left parentheses, guess less than 1, 2 pipes, guess greater than 10, right parentheses. Line 22, indented twice: left brace. Line 23, indented thrice: System dot out dot print l n, left parentheses, double quotes, Well, if you're not going to try, comma, double quotes. Line 24, indented multiple times: plus, double quotes, I'm not playing. double quotes, right parentheses, semicolon. Line 25, indented twice: right brace. Line 26, indented twice: else. Line 27, indented twice: left brace. Line 28, indented thrice: if, left parentheses, guess, 2 equals sign, secret Number, right parentheses. Line 29, indented 4 times: System dot out dot print l n, left parentheses, double quotes, Hoorah. You win, exclamation mark, double quotes, right parentheses, semicolon. Line 30, indented thrice: else. Line 31, indented thrice: left brace. Line 32, indented 4 times: System dot out dot print l n, left parentheses, double quotes, The number was, double quotes, plus, secret Number, right parentheses, semicolon. Line 33: blank. Line 34, indented 4 times: if, left parentheses, Math dot a b s, left parentheses, guess minus secret Number, right parentheses, greater than 3, right parentheses. Line 35, indented 5 times: System dot out dot print l n, left parentheses, double quotes, You missed it by a mile, exclamation mark, double quotes, right parentheses, semicolon. Line 36, indented 4 times: else. Line 37, indented 5 times: System dot out dot print l n, left parentheses, double quotes, You were close, period, double quotes, right parentheses, semicolon. Line 38: blank. Line 39, indented 4 times: System dot out dot print l n, left parentheses, double quotes, Better luck next time, period, double quotes, right parentheses, semicolon. Line 40, indented thrice: right brace. Line 41, indented twice: right brace. Line 42: indented once: right brace. Line 43: right brace.

Back to Example

Program code. The code contains the following keywords: int, if, else. Line 1: System dot out dot print t, left parentheses, double quotes, Enter an integer, right angled bracket, double quotes, right parentheses, semicolon. Line 2: int x equals scan dot next Int, left parentheses, right parentheses, semicolon. Line 3: if, left parentheses, x, right angled bracket, 0, right parentheses. Line 4, indented once: System dot out dot print l n, left parentheses, x plus, double quotes, is positive, double quotes, right parentheses, semicolon. Line 5: else. Line 6, indented once: System dot out dot print l n, left parentheses, x plus double quotes, is negative, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: int, if, else. Line 1: System dot out dot print t, left parentheses, double quotes, Enter an integer, right angled bracket, double quotes, right parentheses, semicolon. Line 2: int x equals scan dot next Int, left parentheses, right parentheses, semicolon. Line 3: if, left parentheses, x, right angled bracket, 0, right parentheses. Line 4, indented once: System dot out dot print l n, left parentheses, x plus, double quotes, is positive, double quotes, right parentheses, semicolon. Line 5: else if, left parentheses, x left angled bracket 0, right parentheses. Line 6, indented once: System dot out dot print l n, left parentheses, x plus double quotes, is negative, double quotes, right parentheses, semicolon. Line 7: else. Line 8, indented once: System dot out dot print l n, left parentheses, double quotes, The integer is 0, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, void, int, String. Line 1: public void work With If Else, left parentheses, int score, right parentheses. Line 2: left brace. Line 3, indented once: String result, equals, double quotes, 3 question marks, double quotes, semicolon. Line 4, indented once: 2 forward slashes, 5 asterisks, Student code starts here. Line 5, indented once: 2 forward slashes. If score is greater than 72, comma, assign, double quotes, over par, double quotes, to result. Line 6, indented once: If score is equal to 72, comma, assign, double quotes, par, double quotes, to result. Line 7, indented once: 2 forward slashes. If score is less than 72, comma, assign, double quotes, below par, double quotes, to result. Lines 8, 9, and 10: blank. Line 11, indented once: 2 forward slashes. Line 12, indented once: 2 forward slashes. Student code ends here. Line 13, indented once: 2 forward slashes. Line 14: blank. Line 15, indented once: animate, left parentheses, score, comma, result, right parentheses, semicolon. Line 16: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, float, double, if, else. Line 1: forward slash, asterisk. Using equality operators on floating hyphen point numbers. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Equality Floating Point. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented once: 2 forward slashes. Part 1, colon, Compute 11, asterisk, decimal point, 1, two ways. Line 10: blank. Line 11, indented once: double d 1, equals, decimal point, 0, semicolon. 2 forward slashes, add, decimal point, 1, to 0 eleven times. Line 12, indented once: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 1. Line 13, indented once: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 2. Line 14, indented once: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 3. Line 15, indented once: d 1, plus, equals, decimal point 1, semicolon. 2 forward slashes, 4. Line 16, indented once: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 5. Line 17, indented once: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 6. Line 18, indented once: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 7. Line 19, indented once: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 8. Line 20, indented once: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 9. Line 21, indented once: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 10. Line 22, indented once: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes. 11. Line 23: blank. Line 24, indented once: double d 2, equals, decimal point 1, asterisk, 11, semicolon. 2 forward slashes, compute 11 asterisk, decimal point 1. Line 25: blank. Line 26, indented once: System dot out dot print l n, left parentheses, double quotes, d 1, equals, double quotes, plus, d 1, right parentheses, semicolon. Line 27, indented once: System dot out dot print l n, left parentheses, double quotes, d 2, equals, double quotes, plus, d 2, right parentheses, semicolon. Line 28, indented once: if, left parentheses, d 1, 2 equals sign, d 2, right parentheses. Line 29, indented twice: System dot out dot print l n, left parentheses, double quotes, d 1 and d 2 are equal, double quotes, right parentheses, semicolon. Line 30, indented once: else. Line 31, indented twice: System dot out dot print l n, left parentheses, double quotes, d 1 and d 2 are not equal, double quotes, right parentheses, semicolon. Line 32: blank. Line 33, indented once: 2 forward slashes. Part 2, colon, Compare float and double with same value. Line 34: blank. Line 35, indented once: float p i F, equals, 3.141592653589793, lowercase f, semicolon. Line 36, indented once: double p i D, equals, 3.141592653589793, semicolon. Line 37: blank. Line 38, indented once: System dot out dot print l n, left parentheses, double quotes, back slash, n, p i F, equals, double quotes, plus, p i F, right parentheses, semicolon. Line 39, indented once: System dot out dot print l n, left parentheses, double quotes, p i d, equals, double quotes, plus, p i D, right parentheses, semicolon. Line 40, indented once: if, left parentheses, p i F, 2 equals sign, p i D, right parentheses. Line 41, indented twice: System dot out dot print l n, left parentheses, double quotes, p i F and p i D are equal, double quotes, right parentheses, semicolon. Line 42, indented once: else. Line 43, indented twice: System dot out dot print l n, left parentheses, double quotes, p i F and p i D are not equal, double quotes, right parentheses, semicolon. Line 44, indented once: right brace. Line 45: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, final, double, if, float, else. Line 1: forward slash, asterisk, Using a threshold to compare floating hyphen point numbers. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: public class Comparing Floating Point. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: final double THRESHOLD, equals, decimal point, 0 0 0 1, semicolon. Line 10: blank. Line 11, indented twice: 2 forward slashes, Part 1, colon, Compute 11, asterisk, decimal point, 1, two ways. Line 12, indented twice: double d 1, equals, decimal point, 0, semicolon. 2 forward slashes, add decimal point, 1, to 0, eleven times. Line 13, indented twice: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 1. Line 14, indented twice: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 2. Line 15, indented once: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 3. Line 16, indented twice: d 1, plus, equals, decimal point 1, semicolon, 2 forward slashes, 4. Line 17, indented twice: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 5. Line 18, indented twice: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 6. Line 19, indented twice: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 7. Line 20, indented twice: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 8. Line 21, indented twice: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 9. Line 22, indented twice: d 1, plus, equals, decimal point, 1, semicolon, 2 forward slashes, 10. Line 23, indented twice: d 1, plus, equals, decimal point, 1, semicolon. 2 forward slashes, 11. Line 24: blank. Line 25, indented twice: double d 2, equals, decimal point, 1, asterisk, 11, semicolon. 2 forward slashes, compute 11, asterisk, decimal point, 1. Line 26: blank. Line 27, indented twice: System dot out dot print l n, left parentheses, double quotes, d 1, equals, double quotes, plus, d 1, right parentheses, semicolon. Line 28, indented twice: System dot out dot print l n, double quotes, d 2, equals, double quotes, plus, d 2, right parentheses, semicolon. Line 29, indented once: if, left parentheses, Math, dot, a b s, left parentheses, d 1, minus, d 2, right parentheses, less than, THRESHOLD, right parentheses. Line 30, indented twice: System dot out dot print l n, left parentheses, double quotes, d 1 and d 2 are considered equal, double quotes, right parentheses, semicolon. Line 31, indented twice:

else. Line 32, indented twice: System dot out dot print l n, left parentheses, double quotes, d 1 and d 2 are not equal, double quotes, right parentheses, semicolon. Line 33: blank. Line 34, indented twice: 2 forward slashes. Part 2, colon, Compare float and double with same value. Line 35, indented twice: float p i F, equals, 3.141592653589793, f, semicolon. Line 36, indented once: double p i D, equals, 3.141592653589793, semicolon. Line 37: blank. Line 38, indented twice: System dot out dot print l n, left parentheses, double quotes, back slash, n, p i F, equals, double quotes, plus, p i F, right parentheses, semicolon. Line 39, indented twice: System dot out dot print l n, left parentheses, double quotes, p i D, equals, double quotes, plus, p i D, right parentheses, semicolon. Line 40, indented twice: if, left parentheses, Math dot a b s, left parentheses, p i F minus p i D, right parentheses, less than, THRESHOLD, right parentheses. Line 41, indented thrice: System dot out dot print l n, left parentheses, double quotes, p i F and p i D are considered equal, double quotes, right parentheses, semicolon. Line 42, indented twice: else. Line 43, indented thrice: System dot out dot print l n, left parentheses, double quotes, p i F and p i are not equal, double quotes, right parentheses, semicolon. Line 44, indented once: right brace. Line 45: right brace.

Back to Example

The Big Decimal Class Constructor Summary is as follows. Big Decimal, left parentheses, String d d d, right

parentheses, creates a Big Decimal object equivalent to the decimal number expressed as a String. The second table is titled Big Decimal Class Method Summary and it has 2 columns labelled Return value and Method name and argument list. The row entries are as follows.
Row 1. Return value: Big Decimal. Method name and argument list: add, left parentheses, Big Decimal n u m, right parentheses, returns a Big Decimal object equal to the current Big Decimal object plus num.
Row 2. Return value: Big Decimal. Method name and argument list: subtract, left parentheses, Big Decimal num, right parentheses, returns a Big Decimal object equal to the current Big Decimal object minus num.
Row 3. Return value: Big Decimal. Method name and argument list: multiply, left parentheses, Big Decimal num, right parentheses, returns a Big Decimal object equal to the current Big Decimal object times num.
Row 4. Return value: Big Decimal. Method name and argument list: divide, left parentheses, Big Decimal num, right parentheses, returns a Big Decimal object equal to the current Big Decimal object divided by num.
Row 5. Return value: int. Method name and argument list: compare To, left parentheses, Big Decimal num, right parentheses, returns 0 if the current Big Decimal object is equal to num; negative 1 if the current Big Decimal object is less than num; and 1 if the current Big Decimal object is greater than num.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, if, else. Line 1: forward slash, asterisk, Using Big Decimal to compute precise decimal numbers. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: import java dot math dot Big Decimal, semicolon. Line 6: blank. Line 7: public class Using Big Decimal. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Big Decimal d 1, equals, new Big Decimal, left parentheses, double quotes, 0.0, double quotes, right parentheses, semicolon. Line 12, indented twice: Big Decimal point One, equals, new Big Decimal, left parentheses, double quotes, 0.1, double quotes, right parentheses, semicolon. Line 13: blank. Line 14, indented twice: 2 forward slashes, Compute 11, asterisk, decimal point, 1, two ways. Line 15, indented twice: 2 forward slashes, add, decimal point, 1, to d 1, eleven times. Line 16, indented twice: d 1, equals, d 1, dot, add, left parentheses, point One, right parentheses, semicolon. 2 forward slashes, 1. Line 17, indented twice: d 1, equals, d 1 dot add, left parentheses, point One, right parentheses, semicolon. 2 forward slashes, 2. Line 18, indented twice: d 1, equals, d 1 dot add, left parentheses, point One, right parentheses, semicolon. 2 forward slashes, 3. Line 19, indented twice: d 1, equals, d 1 dot add, left parentheses, point One, right parentheses, semicolon. 2 forward slashes, 4. Line 20, indented twice: d 1, equals, d 1, dot, add, left parentheses, point One, right parentheses, semicolon. 2 forward slashes, 5. Line 21, indented twice: d 1, equals, d 1 dot add, left parentheses, point One, right parentheses, semicolon. 2 forward slashes, 6. Line 22, indented twice: d 1, equals, d 1 dot add, left parentheses, point One, right parentheses, semicolon. 2 forward slashes, 7. Line 23, indented twice: d 1, equals, d 1 dot add, left parentheses, point One, right parentheses, semicolon. 2 forward slashes, 8. Line 24, indented twice: d 1, equals, d 1, dot, add, left parentheses, point One, right parentheses, semicolon. 2 forward slashes, 9. Line 25, indented twice: d 1, equals, d 1 dot add, left parentheses, point One, right parentheses, semicolon. 2 forward slashes, 10. Line 26, indented twice: d 1, equals, d 1, dot, add, left parentheses, point One, right parentheses, semicolon. 2 forward slashes, 11. Line 27: blank. Line 28, indented twice: 2 forward slashes, multiply, decimal point 1, asterisk, 11. Line 29, indented twice: Big Decimal, d 2, equals, new Big Decimal, left parentheses, double quotes, 0.1, double quotes, right parentheses, semicolon. Line 30, indented twice: Big Decimal, eleven, equals, new, Big Decimal, left parentheses, double quotes, 11, double quotes, right parentheses, semicolon. Line 31, indented twice: d 2, equals, d 2, dot, multiply, left parentheses, eleven, right parentheses, semicolon. Line 32: blank. Line 33, indented twice: System dot out dot print l n, left parentheses, double quotes, d 1, equals, double quotes, plus, d 1, right parentheses, semicolon. Line 34, indented twice: System dot out dot print l n, left parentheses, double quotes, d 2, equals, double quotes, plus, d 2, right parentheses, semicolon. Line 35, indented twice: if, left parentheses, d 1 dot compare To, left parentheses, d 2, right parentheses, 2 equals sign, 0, right parentheses. Line 36, indented thrice: System dot out dot print l n, left parentheses, double quotes, d 1 and d 2 are equal, double quotes, right parentheses, semicolon. Line 37, indented twice: else. Line 38, indented thrice: System dot out dot print l n, left parentheses, double quotes, d 1 and d 2 are not equal, double quotes, right parentheses, semicolon. Line 39, indented once: right brace. Line 40: right brace.

Back to Example

public boolean equals, left parentheses, Object o b, right parentheses.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, new, if, else. Line 1: Forward slash, asterisk, Comparing Object References and data. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: public class Comparing Objects. Line 5: left brace. Line 6, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 7, indented once: left brace. Line 8, indented twice: String s 1, equals, new String, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 9, indented twice: String s 2, equals, new String, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 10: blank. Line 11, indented twice: if, left parentheses, s 1, 2 equals sign, s 2, right parentheses. Line 12, indented thrice: System dot out dot print l n, left parentheses, double quotes, Using, 2 equals sign, comma, s 1 and s 2 are equal, double quotes, right parentheses, semicolon. Line 13, indented twice: else. Line 14, indented thrice: System dot out dot print l n, left parentheses, double quotes, Using, 2 equals sign, comma, s 1 and s 2 are different, double quotes, right parentheses, semicolon. Line 15: blank. Line 16, indented twice: if, left parentheses, s 1 dot equals, left parentheses, s 2, right parentheses, right parentheses. Line 17, indented thrice: System dot out dot print l n, left parentheses, double quotes, Using equals, comma, s 1 and s 2 are equal, double quotes, right parentheses, semicolon. Line 18, indented twice: else. Line 19, indented thrice: System dot out dot print l n, left parentheses, double quotes, Using equals, comma, s 1 and s 2 are different, double quotes, right parentheses, semicolon. Line 20, indented once: right brace. Line 21: right brace.

Back to Example

The table is titled, String Methods for Comparing String

Values. The table has 2 columns labelled Return value and Method name and argument list. The row entries are as follows.
Row 1. Return value: boolean. Method name and argument list: equals, left parentheses, String s t r, right parentheses.
Row 2. Return value: boolean. Method name and argument list: equals Ignore Case, left parentheses, String s t r, right parentheses, compares the value of two Strings, treating upper and lowercase characters as equal. Returns true if the Strings are equal; false otherwise.
Row 3. Return value: int. Method name and argument list: compare To, left parentheses, String s t r, right parentheses, compares the value of the two Strings in lexicographic order. If the String object is less than the String argument, s t r, a negative integer is returned. If the String object is greater than the String argument, a positive number is returned; if the two Strings are equal, 0 is returned.

Back to Table

Program code. The code contains the following keywords: public, class, static, void, String, if, else. Line 1: Forward slash, asterisk. Demonstration of the String comparison methods. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: public class Comparing Strings. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: String title 1, equals, double quotes, Green Pastures, double quotes, semicolon. Line 10, indented twice: String title 2, equals, double quotes, Green Pastures, roman numeral 2, double quotes, semicolon. Line 11, indented twice: String title 3, equals, double quotes, green pastures, double quotes, semicolon. Line 12: blank. Line 13, indented twice: System dot out dot print, left parentheses, double quotes, Using equals, colon, blank space, double quotes, right parentheses, semicolon. Line 14, indented twice: if, left parentheses, title 1 dot equals, left parentheses, title 3, right parentheses, right parentheses. Line 15, indented thrice: System dot out dot print l n, left parentheses, title 1, plus, double quotes, blank space, equals, blank space, double quotes, plus, title 3, right parentheses, semicolon. Line 16, indented twice: else. Line 17, indented thrice: System dot out dot print l n, left parentheses, title 1, plus, double quotes, blank space, is not equal to, blank space, double quotes, plus, title 3, right parentheses, semicolon. Line 18: blank. Line 19, indented twice: System dot out dot print, left parentheses, double quotes, Using equals Ignore Case, colon, blank space, double quotes, right parentheses, semicolon. Line 20, indented twice: if, left parentheses, title 1 dot equals Ignore Case, left parentheses, title 3, right parentheses, right parentheses. Line 21, indented thrice: System dot out dot print l n, left parentheses, title 1, plus, double quotes, blank space, equals, blank space, double quotes, plus, title 3, right parentheses, semicolon. Line 22, indenteed twice: else. Line 23, indented thrice: System dot out dot print l n, left parentheses, title 1, plus, double quotes, blank space, is not equal to, blank space, double quotes, plus, title 3, right parentheses, semicolon. Line 24: blank. Line 25, indented twice: System dot out dot print, left parentheses, double quotes, Using compare To, colon, blank space, double quotes, right parentheses, semicolon. Line 26, indented twice: if, left parentheses, title 1 dot compare To, left parentheses, title 3, right parentheses, greater than 0, right parentheses. Line 27, indented thrice: System dot out dot print l n, left parentheses, title 1, plus, double quotes, blank space, is greater than, blank space, double quotes, plus, title 3, right parentheses, semicolon. Line 28, indented twice: else if, left parentheses, title 1 dot compare To, left parentheses, title 3, right parentheses, less than 0, right parentheses. Line 29, indented thrice: System dot out dot print l n, left parentheses, title 1, plus, double quotes, blank space, is less than, blank space, double quotes, plus, title 3, right parentheses, semicolon. Line 30, indented twice: else. Line 31, indented thrice: System dot out dot print l n, left parentheses, title 1, plus, double quotes, blank space, is equal to, blank space, double quotes, plus, title 3, right parentheses, semicolon. Line 32: blank. Line 33, indented twice: System dot out dot print, left parentheses, double quotes, Using compare To, colon, blank space, double quotes, right parentheses, semicolon. Line 34, indented twice: if, left parentheses, title 1 dot compare To, left parentheses, title 2, right parentheses, greater than 0, right parentheses. Line 35, indented thrice: System dot out dot print l n, left parentheses, title 1, plus, double quotes, blank space, is greater than, blank space, double quotes, plus, title 2, right parentheses, semicolon. Line 36, indented twice: else if, left parentheses, title 1 dot compare To, left parentheses, title 2, right parentheses, less than 0, right parentheses. Line 37, indented thrice: System dot out dot print l n, left parentheses, title 1, plus, double quotes, blank space, is less than, blank space, double quotes, plus, title 2, right parentheses, semicolon. Line 38, indented twice: else. Line 39, indented thrice: System dot out dot print l n, left parentheses, title 1, plus, double quotes, plus, is equal to, blank space, double quotes, plus, title 2, right parentheses, semicolon. Line 42, indented once: right brace. Line 41: right brace.

Back to Example

The syntax is as follows: left parentheses, condition, question mark, expression 1, colon, expression 2, right parentheses.

Back to Code

The conditional statement is as follows. variable equals, left parentheses, condition, question mark, expression 1, colon, expression 2, right parentheses, semicolon.

Back to Code

The equivalent if else statement is as follows. Line 1: if, left parentheses, condition, right parentheses. Line 2, indented once: variable equals expression 1, semicolon. Line 3: else. Line 4, indented once: variable equals expression 2, semicolon.

Back to Code

Program code. The code contains the keywords if and else. Line 1: if, left parentheses, number percentage sign 2, 2 equals sign, 0, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, number plus double quotes, blank space, is even, period, double quotes, right parentheses, semicolon. Line 3: else. Line 4 indented once: System dot out dot print l n, left parentheses, number plus, double quotes, blank space, is odd, period, double quotes, right parentheses.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, String, int. Line 1: Forward slash, asterisk. Using the conditional operator. Line 2, indented twice: Anderson, comma, Franceschi. Line 3: asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Odd Or Even. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line

11, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 12: blank. Line 13, indented twice: System dot out dot print, left parentheses, double quotes, Enter an integer, right angled bracket, double quotes, right parentheses, semicolon. Line 14, indented twice: int number, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 15: blank. Line 16, indented twice: String outcome, equals, left parentheses, number, percentage sign, 2, 2 equals sign, 0, question mark, double quotes, even, period, double quotes, colon, double quotes, odd, period, double quotes, right parentheses, semicolon. Line 17: blank. Line 18, indented twice: System dot out dot print l n, left parentheses, number, plus, double quotes, blank space, is, double quotes, plus, outcome, right parentheses, semicolon. Line 19: blank. Line 20, indented once: right brace. Line 21: right brace.

Back to Example

The table has 3 columns labeled Operation Hierarchy, Order of Same Statement Evaluation, and Operation. Row 1. Operation Hierarchy: left parentheses, right parentheses. Order of Same Statement Evaluation: left to right. Operation: parentheses for explicit grouping. Row 2. Operation Hierarchy: plus plus, minus minus. Order of Same Statement Evaluation: right to left. Operation: shortcut postincrement. Row 3. Operation Hierarchy: plus plus, minus minus, exclamation mark. Order of Same Statement Evaluation: right to left. Operation: shortcut preincrement, logical unary NOT. Row 4. Operation Hierarchy: asterisk, forward slash, percentage sign. Order of Same Statement Evaluation: left to right. Operation: multiplication, division, modulus. Row 5. Operation Hierarchy: plus, minus. Order of Same Statement Evaluation: left to right. Operation: addition or String concatenation, subtraction. Row 6. Operation Hierarchy: less than, less than or equal to, greater than, greater than or equal to. Order of Same Statement Evaluation: left to right. Operation: relational operators: less than, less than or equal to, greater than, greater than or equal to. Row 7. Operation Hierarchy: 2 equals sign, exclamation mark equals. Order of Same Statement Evaluation: left to right. Operation: equality operators: equal to and not equal to. Row 8. Operation Hierarchy: 2 ampersand signs. Order of Same Statement Evaluation: left to right. Operation: logical AND. Row 9. Operation Hierarchy: 2 pipes. Order of Same Statement Evaluation: left to right. Operation: logical OR. Row 10. Operation Hierarchy: question mark, colon. Order of Same Statement Evaluation: left to right. Operation: conditional operator. Row 11. Operation Hierarchy: equals, plus equals, minus equals, asterisk equals, forward slash equals, percentage sign equals. Order of Same Statement Evaluation: right to left. Operation: assignment operator and shortcut assignment operators.

Back to Table

Program code. The code contains the keywords switch, case and default. Line 1: switch, left parentheses, expression, right parentheses. Line 2: left brace. Line 3, indented once: case constant 1 A, comma, constant 1 B, comma, dot, dot, dot, hyphen, right angled bracket, statement or block, semicolon. Line 4, indented once: case constant 2 A, comma, constant 2 B, comma, dot, dot, dot, hyphen, right angled bracket, statement or block, semicolon. Line 5: dot, dot, dot. Line 6, indented once: default, hyphen, right angled bracket, statement or block, semicolon. 2 forward slashes. optional, left parentheses, but often needed, right parentheses. Line 7: right brace.

Back to Code

Program code. The code contains the keywords switch, case, and default. Line 1: switch, left parentheses, letter Grade, right parentheses. Line 2: left brace. Line 3, indented once: case, single quote, uppercase a, single quote, hyphen, right angled bracket, System dot out dot print l n, left parentheses, double quotes, 90 hyphen 100, double quotes, right parentheses, semicolon. Line 4, indented once: case, single quote, uppecase b, single quote, hyphen, right angled bracket, System dot out dot print l n, left parentheses, double quotes, 80 hyphen 89, double quotes, right parentheses, semicolon. Line 5, indented once: default, hyphen, right angled bracket, System dot out dot print l n, left parentheses, double quotes, 0 hyphen 79, double quotes, right parentheses, semicolon. Line 6: right brace.

Back to Code

Program code. The code contains the keywords if, and else. Line 1: if, left parentheses, letter Grade, 2 equals sign, single quote, uppercase a, single quote, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, 90 hyphen 100, double quotes, right parentheses, semicolon. Line 3: else if, left parentheses, letter Grade, 2 equals sign, single quotes, uppercase b, single quote, right parentheses. Line 4, indented once: System dot out dot print l n, left parentheses, double quotes, 80 hyphen 89, double quotes, right parentheses, semicolon. Line 5: else. Line 6, indented once: System dot out dot print l n, left parentheses, double quotes, 0 hyphen 79, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the keywords switch, case, break, and default. Line 1: switch, left parentheses, letter Grade, right parentheses. Line 2: left brace. Line 3, indented once: case, single quote, uppercase a, single quote, colon. Line 4, indented twice: System dot out dot print l n, left parentheses, double quotes, 90 hyphen 100, double quotes, right parentheses, semicolon. Line 5, indented twice: break, semicolon. Line 6, indented once: case, single quote, uppercase b, single quote, colon. Line 7, indented twice: System dot out dot print l n, left parentheses, double quotes, 80 hyphen 89, double quotes, right parentheses, semicolon. Line 8, indented twice: break, semicolon. Line 9, indented once: default, colon. Line 10, indented twice: System dot out dot print l n, left parentheses, double quotes, 0 hyphen 79, double quotes, right parentheses, semicolon. Line 11: right brace.

Back to Code

Program code. The code contains the keywords switch, case, yield and default. Line 1: data type identifier equals switch, left parentheses, expression, right parentheses. Line 2: left brace. Line 3, indented once: case constant 1 A, comma, constant 1 B, comma, dot, dot, dot, hyphen, right angled bracket, expression or block with yield, semicolon. Line 4, indented once: case constant 2 A, comma, constant 2 B, comma, dot, dot, dot, hyphen, right angled bracket, expression or block with yield, semicolon. Line 5: dot, dot, dot. Line 6, indented once: 2 forward slashes, optional default, left parentheses, but often needed, right parentheses. Line 7, indented once: default, hyphen, right angled bracket, expression or block with yield, semicolon. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, new, double, String, swtich, case, if, else, yield, default. Line 1: Forward slash, asterisk. A simple calculator. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: import java dot text dot Decimal Format, semicolon. Line 6: import java dot util dot Scanner, semicolon. Line 7: blank. Line 8: public class Calculator. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 13: blank. Line 14, indented twice: 2 forward slashes. set up the output format of the result. Line 15, indented twice: Decimal Format two Decimals, equals, new Decimal Format, left parentheses, double quotes, hash, comma, hash, hash, hash, decimal point, hash, hash, double quotes, right parentheses, semicolon. Line 16: blank. Line 17, indented twice: 2 forward slashes. print a welcome message. Line 18, indented twice: System dot out dot print l n, left parentheses, double quotes, Welcome to the Calculator, double quotes, right parentheses, semicolon. Line 19: blank. Line 20, indented twice: 2 forward slashes, define and print a menu, comma, then prompt for the operation. Line 21, indented twice: String menu, equals, 3 double quotes. Line 22, indented thrice: Operations are, colon. Line 23, indented thrice: backslash t, ADD, or, plus, for addition. Line 24, indented thrice: backslash t, SUBTRACT or minus for subtraction. Line 25, indented thrice: backslash t, MULTIPLY or asterisk for multiplication. Line 26, indented thrice: backslash t, DIVIDE of forward slash for division, 3 double quotes, semicolon. Line 27, indented twice: System dot out dot print l n, left parentheses, menu, right parentheses, semicolon. Line 28, indented twice: System dot out dot print, left parentheses, double quotes, Enter your operator, colon, blank space, double quotes, right parentheses, semicolon. Line 29, indented twice: String operation, equals, scan dot next, left parentheses, right parentheses, semicolon. Line 30, indented twice: operation, equals, operation dot to Upper Case, left parentheses, right parentheses, semicolon. Line 31: blank. Line 32, indented twice: 2 forward slashes, read the two operands. Line 33, indented twice: System dot out dot print, left parentheses, double quotes, Enter the first operand, colon, blank space, double quotes, right parentheses, semicolon. Line 34, indented twice: double f p 1, equals, scan dot next Double, left parentheses, right parentheses, semicolon. Line 30, indented twice: operation, equals, operation dot to Upper Case, left parentheses, right parentheses, semicolon. Line 31: blank. Line 32, indented twice: 2 forward slashes. read the two operands. Line 33, indented twice: System dot out dot print, left parentheses, double quotes, Enter the first operand, colon, blank space, double quotes, right parentheses, semicolon. Line 34, indented twice: double f p 1, equals, scan dot next Double, left parentheses, right parentheses, semicolon. Line 35, indented twice: System dot out dot print, left parentheses, double quotes, Enter the second operand, colon, blank space, double quotes, right parentheses, semicolon. Line 36, indented twice:

double f p 2, equals, scan dot next Double, left parentheses, right parentheses, semicolon. Line 37: blank. Line 38, indented twice: 2 forward slashes. perform the operation. Line 39, indented twice: String result, equals, switch, left parentheses, operation, right parentheses. Line 40, indented twice: left brace. Line 41, indented thrice: case, double quotes, ADD, double quotes, comma, double quotes, plus, double quotes, hyphen, right angled bracket, two Decimals dot format, left parentheses, f p 1, plus, f p 2, right parentheses, semicolon. Line 42, indented thrice: case, double quotes, SUBTRACT, double quotes, comma, double quotes, minus sign, double quotes, hyphen, right angled bracket, two Decimals dot format, left parentheses, f p 1 minus f p 2, right parentheses, semicolon. Line 43, indented thrice: case double quotes, MULTIPLY, double quotes, comma, double quotes, asterisk, double quotes, hyphen, right angled bracket, two Decimals dot format, left parentheses, f p 1 asterisk f p 2, right parentheses, semicolon. Line 44, indented thrice: case, double quotes, DIVIDE, double quotes, comma, double quotes, forward slash, double quotes, hyphen, right angled bracket. Line 45, indented thrice: left brace. Line 46, indented 4 times: if, left parentheses, f p 2, 2 equals sign, 0.0, right parentheses. Line 47, indented 5 times: yield, double quotes, Dividing by 0 is not allowed, double quotes, semicolon. Line 48, indented 4 times: else. Line 49, indented 5 times: yield, two Decimals dot format, left parentheses, f p 1, forward slash, f p 2, right parentheses, semicolon. Line 50, indented thrice: right brace. Line 51, indented thrice: default, hyphen, right angled bracket, double quotes, Invalid operation, double quotes, semicolon. Line 52, indented twice: right brace, semicolon. Line 53: blank. Line 54, indented twice: 2 forward slashes. output the result of the operation. Line 55, indented twice: System dot out dot print l n, left parentheses, double quotes, The result is, colon, blank space, double quotes, plus, result, right parentheses, semicolon. Line 56, indented once: right bace. Line 57: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, at symbol Override, void, final, int, switch, case, default, static, void. Line 1: Forward slash, asterisk. Drawing a smiley face. Line 2, indented once: Anderson, comma, Franceschi. Line 3: asterisk, forward slash. Line 4: blank. Line 5: import java f x dot application dot Application, semicolon. Line 6: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 7: import java f x dot scene dot paint dot Color, semicolon. Line 8: import java f x dot scene dot shape dot Arc Type, semicolon. Line 9: import java f x dot stage dot Stage, semicolon. Line 10: blank. Line 11: import java dot util dot Scanner, semicolon. Line 12: blank. Line 13: public class Smiley Face Draw extends Application. Line 14: left brace. Line 15, indented once: at symbol, Override. Line 16, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 17, indented once: left brace. Line 18, indented twice: 2 forward slashes, set up window title and size. Line 19, indented twice: Graphics Context g c, equals, J I Graphics Utility dot set Up Graphics, left parentheses. Line 20, indented multiple times: stage, comma, double quotes, Draw a Smiley Face, double quotes, comma, 400, comma, 400, right parentheses, semicolon. Line 21: blank. Line 22, indented twice: final int, uppercase x, equals, 100, comma, uppercase y, equals, 100, comma, RADIUS, equals, 100, semicolon. Line 23: blank. Line 24, indented twice: 2 forward slashes. ask the user for a rating. Line 25, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 26, indented twice: System dot out dot print, left parentheses, double quotes, Enter a rating, left parentheses, 1 to 5, right parentheses, right angled bracket, double quotes, right parentheses, semicolon. Line 27, indented twice: int rating, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 28: blank. Line 29, indented twice: 2 forward slashes. Draw the face and the eyes. Line 30, indented twice: g c dot set Fill, left parentheses, Color dot YELLOW, right parentheses, semicolon. Line 31, indented twice: g c dot fill Oval, left parentheses, uppercase x, comma, uppercase y, comma, 2 asterisk RADIUS, comma, 2 asterisk RADIUS, right parentheses, semicolon. Line 32, indented twice: g c dot set Fill, left parentheses, Color dot BLACK, right parentheses, semicolon. Line 33, indented twice: g c dot fill Oval, left parentheses, uppercase x, plus, RADIUS, forward slash, 3, comma, uppercase y plus RADIUS, forward slash, 3, comma. Line 34, indented multiple times: RADIUS, forward slash, 3, comma, RADIUS, forward slash, 3, right parentheses, semicolon. Line 35, indented twice: g c dot fill Oval, left parentheses, uppercase x plus 4 asterisk RADIUS forward slash 3, comma, uppercase y plus RADIUS, forward slash, 3, comma. Line 36, indented multiple times: RADIUS, forward slash, 3, comma, RADIUS, forward slash, 3, right parentheses, semicolon. Line 37: blank. Line 38, indented twice: 2 forward slashes, Draw the smile. Line 39, indented twice: switch, left parentheses, rating, right parentheses. Line 40, indented twice: left brace. Line 41, indented thrice: case 1, comma, 2, hyphen, right angled bracket. Line 42, indented 4 times: g c dot stroke Arc, left parentheses, uppercase x plus RADIUS, forward slash, 2, comma, uppercase y, plus, RADIUS, comma. Line 43, indented multiple times: RADIUS, comma, RADIUS forward slash 2, comma, 0, comma, 180, comma, Arc Type dot OPEN, right parentheses, semicolon. Line 44, indented thrice: case 4, comma, 5, hyphen, right angled bracket. Line 45, indented 4 times: g c dot stroke Arc, left parentheses, uppercase x plus RADIUS forward slash 2, comma, uppercase y plus RADIUS, comma. Line 46, indented multiple times: RADIUS, comma, RADIUS, forward slash, 2, comma, 0, comma, negative 180, comma, Arc Type dot OPEN, right parentheses, semicolon. Line 47, indented thrice: default, hyphen, right angled bracket. Line 48, indented 4 times: g c dot stroke Line, left parentheses, uppercase x plus RADIUS forward slash 2, comma, uppercase y plus 4 asterisk RADIUS forward slash 3, comma. Line 49, indented multiple times: uppercase x plus 3 asterisk RADIUS forward slash 2, comma, uppercase y plus 4 asterisk RADIUS forward slash 3, right parentheses, semicolon. Line 50, indented twice: right brace. 2 forward slashes, end switch. Line 41, indented once: right brace. Line 52: blank. Line 53, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 54, indented once: left brace. Line 55, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 56, indented once: right brace. Line 57: right brace.

Back to Example

Program code. The code contains the following keywords: public, void, int. Line 1: 2 forward slashes, 5 asterisks, 1 student writes this method. Line 2: public void work With Switch, left parentheses, int value, right parentheses. Line 3: left brace. Line 4, indented once: 2 forward slashes. Line 5, indented once: 2 forward slashes, Student code starts here. Line 6, indented once: 2 forward slashes. Line 7: blank. Line 8: blank. Line 9, indented once: 2 forward slashes. Line 10, indented once: 2 forward slashes, Student code ends here. Line 11, indented once: 2 forward slashes. Line 12: blank. Line 13: blank. Line 14:

right brace. Line 15: 2 forward slashes, end of work With Switch.

Back to Example

void animate, left parentheses, int case Constant, comma, int value, right parentheses.

Back to Code

animate, left parentheses, 2, comma, value, right parentheses, semicolon.

Back to Code

animate, left parentheses, negative 1, comma, value, right parentheses, semicolon.

Back to Code

Line 1: int a equals 3, semicolon. Line 2: int b equals 5, semicolon.

Back to Code

Line 1: boolean a equals true, semicolon. Line 2: boolean b equals true, semicolon. Line 3: boolean c equals false, semicolon.

Back to Code

Line 1: int a equals 3, semicolon. Line 2: int b equals 5, semicolon.

Back to Code

Line 1: boolean a equals true, semicolon. Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: boolean a equals left parentheses, true, 2 ampersand signs, false, right parentheses, semicolon. Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

Line 1: if, left parentheses, left parentheses, true, 2 pipes, false, right parentheses, 2 ampersands, left parentheses, false, 2 pipes, true, right parentheses, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, Inside true block, double quotes, right parentheses, semicolon. Line 3: System dot out dot print l n, left parentheses, double quotes, End of sequence, double quotes, right parentheses, semicolon.

Back to Code

Line 1: if, left parentheses, 27 percentage sign 3, 2 equals sign, 0, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, 27 is divisible by 3, double quotes, right parentheses, semicolon. Line 3: else. Line 4, indented once: System dot out dot print l n, left parentheses, double quotes, 27 is not divisible by 3, double quotes, right parentheses, semicolon. Line 5: System dot out dot print l n, left parentheses, double quotes, End of sequence, double quotes, right parentheses, semicolon.

Back to Code

Line 1: String s equals double quotes, Hello, double quotes, semicolon. Line 2: if, left parentheses, s dot equals, left parentheses, double quotes, hello, double quotes, right parentheses, right parentheses. Line 3, indented once: System dot out dot print l n, left parentheses, double quots, String is hello, double quotes, right parentheses, semicolon. Line 4: else. Line 5, indented once: System dot out dot print l n, left parentheses, double quotes, String is not hello, double quotes, right parentheses, semicolon. Line 6: System dot out dot print l n, left parentheses, double quotes, End of sequence, double quotes, right parentheses, semicolon.

Back to Code

Line 1: int grade equals 77, semicolon. Line 2: if, left parentheses, grade greater than or equal to 90, right parentheses. Line 3, indented once: System dot out dot print l n, left parentheses, double quotes, uppercase a, double quotes, right parentheses, semicolon. Line 4: else if, left parentheses, grade greater than or equal to 80, right parentheses. Line 5, indented once: System dot out dot print l n, left parentheses, double quotes, uppercase b, double quotes, right parentheses, semicolon. Line 6: else if, left parentheses, grade greater than or equal to 70, right parentheses. Line 7, indented once: System dot out dot print l n, left parentheses, double quotes, uppercase c, double quotes, right parentheses, semicolon. Line 8: else. Line 9, indented once: System dot out dot print l n, left parentheses, double quotes, uppercase d or lower, double quotes, right parentheses, semicolon. Line 10: System dot out dot print l n, left parentheses, double quotes, Done, double quotes, right parentheses, semicolon.

Back to Code

Line 1: int a equals 65, semicolon. Line 2: boolean b equals false, semicolon. Line 3: blank. Line 4: if, left parentheses, a greater than or equal to 70, right parentheses. Line 5: left brace. Line 6, indented once: System dot out dot print l n, left parentheses, double quotes, Hello 1, double quotes, right parentheses, semicolon. Line 7, indented once: if, left parentheses, b, 2 equals sign, true, right parentheses. Line 8, indented twice: System dot out dot print l n, left parentheses, double quotes, Hello 2, double quotes, right parentheses, semicolon. Line 9: right brace. Line 10: else. Line 11: left brace. Line 12, indented once: System dot out dot print l n, left parentheses, double quotes, Hello 3, double quotes, right parentheses, semicolon. Line 13, indented once: if, left parentheses, b, 2 equals sign, false, right parentheses. Line 14, indented twice: System dot out dot print l n, left parentheses, double quotes, Hello 4, double quotes, right parentheses, semicolon. Line 15: right brace. Line 16: System dot out dot print l n, left parentheses, double quotes, Done, double quotes, right parentheses, semicolon.

Back to Code

Line 1: int season equals 3, semicolon. Line 2: switch, left parentheses, season, right parentheses. Line 3: left brace. Line 4, indented once: case 1, hyphen, right angled bracket. Line 5, indented twice: System dot out dot print l n, left parentheses, double quotes, Season is Winter, double quotes, right parentheses, semicolon. Line 6, indented once: case 2, hyphen, right angled bracket. Line 7, indented twice: System dot out dot print l n, left parentheses, double quotes, Season is Spring, double quotes, right parentheses, semicolon. Line 8, indented once: case 3, hyphen, right angled bracket. Line 9, indented twice: System dot out dot print l n, left parentheses, double quotes, Season is Summer, double quotes, right parentheses, semicolon. Line 10, indented once: case 4, hyphen, right angled bracket. Line 5, indented twice: System dot out dot print l n, left parentheses, double quotes, Season is Fall, double quotes, right parentheses, semicolon. Line 6, indented once: default, hyphen, right angled bracket. Line 7, indented twice: System dot out dot print l n, left parentheses, double quotes, Invalid season, double quotes, right parentheses, semicolon. Line 8: right brace.

Back to Code

Line 1: char c equals, single quote, e, single quote, semicolon. Line 2: switch, left parentheses, c, right parentheses. Line 3: left brace. Line 4, indented once: case, single quote, uppercase h, single quote, hyphen, right angled bracket. Line 5, indented twice: System dot out dot print l n, left parentheses, double quotes, letter 1, double quotes, right parentheses, semicolon. Line 6, indented once: case, single quote, lowercase e, single quote, hyphen, right angled bracket. Line 7, indented twice: System dot out dot print l n, left parentheses, double quotes, letter 2, double quotes, right parentheses, semicolon. Line 8, indented once: case, single quote, lowercase l, single quote, hyphen, right angled bracket. Line 9, indented twice: System dot out dot print l n, left parentheses, double quotes, letters 3 and 4, double quotes, right parentheses, semicolon. Line 10, indented once: case, single quote, lowercase o, single quote, hyphen, right angled bracket. Line 11, indented twice: System dot out dot print l n, left parentheses, double quotes, letter 5, double quotes, right parentheses, semicolon. Line 12, indented once: default, hyphen, right angled bracket. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, letter is not in Hello, double quotes, right parentheses, semicolon. Line 14: right brace.

Back to Code

left parentheses, b 2, 2 equals sign, b 1, right parentheses, AND, left parentheses, a 1, left angled bracket, equals sign, a 2, right parentheses.

Back to Code

Line 1: if, left parentheses, a 1, 2 equals sign, 4, right parentheses, semicolon. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, a 1 equals 4, double quotes, right parentheses, semicolon.

Back to Code

Line 1: boolean b 1 equals true, semicolon. Line 2: if b 1. Line 3, indented once: System dot out dot print l n, left parentheses, double quotes, b 1 is true, double quotes, right parentheses, semicolon.

Back to Code

Line 1: if, left brace, b 2, 2 equals sign, true, right brace. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, b 2 is true, double quotes, right parentheses, semicolon.

Back to Code

Line 1: if, left parentheses, b 1, 2 equals sign, true, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, b 1 is true, double quotes, right parentheses, semicolon. Line 3: else. Line 4, indented once: System dot out dot print l n, left parentheses, double quotes, b 1 is false, double quotes, right parentheses, semicolon. Line 5: else if, left parentheses, a 1, left angled bracket, 100, right parentheses. Line 6, indented once: System dot out dot print l n, left parentheses, double quotes, a 1 is, left angled bracket, equals sign, 100, double quotes, right parentheses, semicolon.

Back to Code

Line 1: if, left parentheses, b 2, 2 equals sign, b 1, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, b 2 and b 1 have the same value, double quotes, right parentheses, semicolon. Line 3: else if, left parentheses, a 1, 2 equals sign, a 2, right parentheses. Line 4, indented once: System dot out dot print l n, left parentheses, double quotes, a 1 and a 2 have the same value, double quotes, right parentheses, semicolon. Line 5: else. Line 6, indented once: System dot out dot print l n, left parentheses, double quotes, All variables are different, double quotes, right parentheses, semicolon.

Back to Code

Line 1: if, left parentheses, b 2, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, b 2 is true, double quotes, right parentheses, semicolon. Line 3: else if, left parentheses, a 1, left angled bracket, equals sign, 10, 2 pipes, a 2, right angled bracket, 50, right parentheses. Line 4: left brace. Line 5, indented once: System dot out dot print l n, left parentheses, double quotes, a 1, left angled bracket, equals sign, 10 or, double quotes, right parentheses, semicolon. Line 6, indented once: System dot out dot print l n, left parentheses, double quotes, a 2, right angled bracket, 50, double quotes, right parentheses, semicolon. Line 7: right brace. Line 8: else. Line 9, indented once: System dot out dot print l n, left parentheses, double quotes, none of the above, double quotes, right parentheses, semicolon.

Back to Code

The code sequence is as follows. Line 1: boolean b equals true, semicolon. Line 2: if, left parentheses, b, right parentheses. Line 3, indented once: System dot out dot print l n, left parentheses, double quotes, Inside true block, double quotes, right parentheses, semicolon. Line 4, indented once: System dot out dot print l n, left parentheses, double quotes, b was true, double quotes, right parentheses, semicolon. Line 5: else. 2 forward slashes. line 12. Line 6, indented once: System dot out dot print l n, left parentheses, double quotes, Inside false block, double quotes, right parentheses, semicolon.

Back to Code

The compile time error is as follows. Line 1: Test dot java, colon, 12, colon, error, colon, single quote, else, single quote, without, single quote, if, single quote. Line 2: else, 2 forward slashes, line 12. Line 3: a caret sign under e of else. Line 4: 1 error.

Back to Code

Teh code sequence is as follows. Line 1: int a equals 32, semicolon. Line 2: if, left parentheses, a equals 31, right parentheses, 2 forward slashes, line 9. Line 3, indented once: System dot out dot print l n, left parentheses, double quotes, The value of a is 31, double quotes, right parentheses, semicolon. Line 4: else. Line 5, indented once: System dot out dot print l n, left parentheses, double quotes, The value of a is not 31, double quotes, right parentheses, semicolon.

Back to Code

The compile time error is as follows. Line 1: Test dot java, colon, 9, colon, error, colon, incompatible types. Line 2: if, left parentheses, a equals 32, right parentheses, 2 forward slashes, line 9. Line 3: a caret sign under the equals sign of line 2. Line 4: 1 error.

Back to Code

The code sequence is as follows. Line 1: boolean b equals true, semicolon. Line 2: if, left parentheses, b, right parentheses. Line 3: left brace. Line 4, indented once:

System dot out dot print l n, left parentheses, double quotes, Inside true block, double quotes, right parentheses, semicolon. Line 5, indented once: System dot out dot print l n, left parentheses, double quotes, b was true, double quotes, right parentheses, semicolon. Line 6: else, 2 forward slashes, line 13. Line 7, indented once: System dot out dot print l n, left parentheses, double quotes, Inside false block, double quotes, right parentheses, semicolon. Line 8: right brace. Line 9: System dot out dot print l n, left parentheses, double quotes, Done, double quotes, right parentheses, semicolon.

Back to Code

The compile time error is as follows. Line 1: Test dot java, colon, 13, colon, error, colon, single quote, else, single quote, without, single quote, if, single quote, period. Line 2: else, 2 forward slashes, line 13. Line 3: a caret sign under e of else. Line 4: 1 error.

Back to Code

Line 1: int b equals 44, semicolon. Line 2: if, left parentheses, b equals 23, right parentheses. Line 3, indented once: System dot out dot print l n, left parentheses, double quotes, Inside true block, double quotes, right parentheses, semicolon.

Back to Code

Line 1: Test dot java, colon, 9, colon, error, colon, incompatible types. Line 2: if, left parentheses, b equals 23, right parentheses. Line 3: A caret sign under the equals sign in Line 2. Line 4: required boolean. Line 5: found, colon, int. Line 6: 1 error.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, String, final, while. Line 1: Forward slash, asterisk. Working with a sentinel value. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: import java dot util dot Scanner, semicolon. Line 5: blank. Line 6: public class Echo User Input. Line 7: left brace. Line 8, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: final int SENTINEL, equals, negative 1, semicolon. Line 11, indented twice: int number, semicolon. Line 12: blank. Line 13, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: 2 forward slashes. priming read. Line 16, indented twice: Syste dot out dot print, left parentheses, double quotes, Enter an integer, comma, or negative 1 to stop, right angled bracket, blank space, double quotes, right parentheses, semicolon. Line 17, indented twice: number, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 18:

blank. Line 19, indented twice: while, left parentheses, number, exclamation mark, equals, SENTINEL, right parentheses. Line 20, indented twice: left brace. Line 21, indented thrice: 2 forward slashes. processing. Line 22, indented thrice: System dot out dot print l n, left parentheses, number, right parentheses, semicolon. Line 23: blank. Line 24, indented thrice: 2 forward slashes. update read. Line 25, indented thrice: System dot out dot print, left parentheses, double quotes, Enter an integer, comma, or negative 1 to stop, right angled bracket, double quotes, right parentheses, semicolon. Line 26, indented thrice: number, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 23: blank. Line 24, indented thrice: 2 forward slashes. update read. Line 25, indented thrice: System dot out dot print, left parentheses, double quotes, Enter an integer, comma, or negative 1 to stop, right angled bracket, double quotes, right parentheses, semicolon. Line 26, indented thrice: number, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 27, indented twice: right brace. Line 28: blank. Line 29, indented twice: System dot out dot print l n, left parentheses, double quotes, Sentinel value detected. Goodbye, double quotes, right parentheses, semicolon. Line 30, indented once: right brace. Line 31: right brace.

Back to Example

The title of the table is, Selected Methods of the Scanner Class. The syntax for the Scanner constructor is as follows: Scanner, left parentheses, File, blank space, file, right parentheses. The description below the syntax is as follows: creates a Scanner object and associates it with a file. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: boolean. Method name and argument list: has Next, left parentheses, right parentheses, returns true if there is another token in the input stream; false, otherwise. Row 2. Return value: byte. Method name and argument list: next Byte, left parentheses, right parentheses, returns the next input as a byte. Row 3. Return value: short. Method name and argument list: next Short, left parentheses, right parentheses, returns the next input as a short. Row 4. Return value: int. Method name and argument list: next Int, left parentheses, right parentheses, returns the next input as an int. Row 5. Return value: long. Method name and argument list: next Long, left parentheses, right parentheses, returns the next input as a long. Row 6. Return value: float. Method name and argument list: next Float, left parentheses, right parentheses, returns the next input as a float. Row 7. Return value: double. Method name and argument list: next Double, left parentheses, right parentheses, returns the next input as a double. Row 8. Return value: boolean. Method name and argument list: next Boolean, left parentheses, right parentheses, returns the next input as a boolean. Row 9. Return value: String. Method name and argument list: next, left parentheses, right parentheses, returns the next token in the input line as a String.

Back to Table

The title of the table is, A constructor for the File class. The syntax for the File class constructor is as follows: File, left parentheses, String path name, right parentheses, constructs a File object with the path name file name so that the file name is platform independent.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, String, throws, int, while. Line 1: Forward slash, asterisk. Reading a Text File. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: import java dot util dot Scanner, semicolon. Line 5: import java dot i o dot File, semicolon. Line 6: import java dot i o dot I O Exception, semicolon. Line 7: blank. Line 8: public class Echo File Data. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses, throws I O Exception. Line 11, indented once: left brace. Line 12, indented twice: int number, semicolon. Line 13: blank. Line 14, indented twice: File input File, equals, new File, left parentheses, double quotes, input dot t x t, double quotes, right parentheses, semicolon. Line 15, indented twice: Scanner file, equals, new Scanner, left parentheses, input File, right parentheses, semicolon.

Line 16: blank. Line 17, indented twice: while, left parentheses, file dot has Next, right parentheses, right parentheses, right parentheses. Line 18, indented twice: left brace. Line 19, indented thrice: 2 forward salshes. read next integer. Line 20, indented thrice: number, equals, file dot next Int, left parentheses, right parentheses, semicolon. Line 21, indented thrice: 2 forward slashes. process the value read. Line 22, indented thrice: System dot out dot print l n, left parentheses, number, right parentheses, semicolon. Line 23, indented twice: right brace. Line 24: blank. Line 25, indented twice: System dot out dot print l n, left parentheses, double quotes, End of file detected. Goodbye, double quotes, right parentheses, semicolon. Line 26, indented once: right brace. Line 27: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, final, int, new, while. Line 1: Forward slash, asterisk. Addition Calculator. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: import. Line 5: import java dot util dot scanner, semicolon. Line 6: blank. Line 7: public class Calculator. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: final int SENTINEL, equals, 0, semicolon. Line 12, indented twice: int number, semicolon. Line 13, indented twice: int total, equals, 0, semicolon. Line 14: blank. Line 15, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 16: blank. Line 17, indented twice: System dot out dot print l n, left parentheses, double quotes, Welcome to the addition calculator, period, backslash, n, double quotes, right parentheses, semicolon. Line 18: blank. Line 19, indented twice: System dot out dot print, left parentheses, double quotes, Enter the first number, double quotes. Line 20, indented multiple times: plus, double quotes, or 0 for the total, right angled bracket, double quotes, right parentheses, semicolon. Line 21, indented twice: number, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 22: blank. Line 23, indented twice: while, left parentheses, number, exclamation mark, equals, SENTINEL, right parentheses. Line 24, indented twice: left brace. Line 25, indented thrice: total, plus, equals, number, semicolon. Line 26: blank. Line 27, indented thrice: System dot out dot print, left parentheses, double quotes, Enter the next number, double quotes. Line 28, indented multiple times: plus, double quotes, or 0 for the total, right angled bracket, double quotes, right parentheses, semicolon. Line 29, indented thrice: number, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 30, indented twice: right brace. Line 31: blank. Line 32, indented twice: System dot out dot print l n, left parentheses, double quotes, The total is, blank space, double quotes, plus, total, right parentheses, semicolon. Line 33, indented once: right brace. Line 34: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, int, final, new, while, if. Line 1: Forward slash. Asterisk. Counting passing test scores. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: import java dot text dot Decimal Format, semicolon. Line 7: blank. Line 8: public class Count Test Scores. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: int count Passed, equals, 0, semicolon. Line 13, indented twice: int count Scores, equals, 0, semicolon. Line 14, indented twice: int score, semicolon. Line 15, indented twice: final int SENTINEL, equals, negative 1, semicolon. Line 16: blank. Line 17, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 18: blank. Line 19, indented twice: System dot out dot print l n, left parentheses, double quotes, This program counts, blank space, double quotes. Line 20, indented multiple times: plus, double quotes, the number of passing test scores, period, double quotes, right parentheses, semicolon. Line 21, indented twice: System dot out dot print l n, left parentheses, double quotes, Enter a minus 1 to stop, period, double quotes, right parentheses, semicolon. Line 22: blank. Line 23, indented twice: System dot out dot print, left parentheses, double quotes, Enter the first score, right angled bracket, double quotes, right parentheses, semicolon. Line 24, indented twice: score, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 25: blank. Line 26, indented twice: while, left parentheses, score, exclamation mark, equals, SENTINEL, right parentheses. Line 27, indented twice: left brace. Line 28, indented thrice: if, left parentheses, score greater than or equal to 60, right parentheses. Line 29, indented thrice: left brace. Line 30, indented 4 times: count Passed, plus, plus, semicolon. Line 31, indented thrice: right brace. Lien 32: blank. Line 33, indented thrice: count Scores, plus, plus, semicolon. Line 34: blank. Line 35, indented thrice: System dot out dot print, left parentheses, double quotes, Enter the next score, right angled bracket, double quotes, right parentheses, semicolon. Line 36, indented thrice: score, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 37, indented twice: right brace. Line 38: blank. Line 39, indented twice: System dot out out print l n, left parentheses, double quotes, You entered, blank space, double quotes, plus, count Scores, plus, double quotes, blank space, scores, double quotes, right parentheses, semicolon. Line 40, indented twice: System dot out dot print l n, left parentheses, double quotes, The number of passing test scores is, blank space, double quotes. Line 41, indented multiple times: plus, count Passed, right parentheses, semicolon. Line 42, indented twice: if, left parentheses, count Scores, exclamation mark, equals, 0, right parentheses. Line 43, indented twice: left brace. Line 44, indented thrice: Decimal Format percent, equals, new, Decimal Format, left parentheses, double quotes, 0.0 percent, double quotes, right parentheses, semicolon. Line 45, indented thrice: System dot out dot print l n, left parentheses. Line 46, indented 4 times: percent dot format, left parentheses, left parentheses, double, right parentheses, count Passed, forward slash, count Scores, right parentheses. Line 47, indented 4 times: plus, double quotes, of the class passed the test, double quotes, right parentheses, semicolon. Line 48, indented twice: right brace. Line 49, indented once: right brace. Line 50: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, int, final, new, while, if, else, double. Line 1: Forward slash. Asterisk. Calculate the average test score. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: import java dot text dot Decimal Format, semicolon. Line 7: blank. Line 8: public class Average Test Score. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: int count, equals, 0, semicolon. Line 13, indented twice: int total equals 0, semicolon. Line 14, indented twice: final int SENTINEL equals negative 1, semicolon. Line 15, indented twice: int score, semicolon. Line 16: blank. Line 17, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 18: blank. Line 19, indented twice: System dot out dot print l n, left parentheses, double quotes, To calculate a class average, comma, double quotes, right parentheses, semicolon. Line 20, indented twice: System dot out dot print l n, left parentheses, double quotes, enter each test score, period, double quotes, right parentheses, semicolon. Line 21, indented twice: System dot out dot print l n, left parentheses, double quotes, When you are finished, comma, enter a minus 1, double quotes, right parentheses, semicolon. Line 22: blank. Line 23, indented twice: System dot out dot print, left parentheses, double quotes, Enter the first test score, right angled bracket, double quotes, right parentheses, semicolon. Line 24, indented twice: score, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 25: blank. Line 26, indented twice: while, left parentheses, score, exclamation mark, equals, SENTINEL, right parentheses. Line 27, indented twice: left brace. Line 28, indented thrice: total, plus, equals, score, semicolon. 2 forward slashes, add score to total. Line 29, indented thrice: count, plus, plus, semicolon. 2 forward slashes, add 1 to count of test scores. Line 30: blank. Line 31, indented thrice: System dot out dot print, left parentheses, double quotes, Enter the next test score, right angled bracket, double quotes, right parentheses, semicolon. Line 32, indented thrice: score, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 33, indented twice: right brace. Line 34: blank. Line 35, indented twice: if, left parentheses, count, exclamation mark, equals, 0, right parentheses. Line 36, indented twice: left brace. Line 37, indented thrice: Decimal Format one Decimal Place, equals, new Decimal Format, left parentheses, double quotes, 0.0, double quotes, right parentheses, semicolon. Line 38, indented thrice: System dot out dot print l n, left parentheses, double quotes, backslash n, The class average is, blank space, double quotes. Line 39, indented multiple times: plus, one Decimal Place dot format, left parentheses, left parentheses, double, right parentheses, total, forward slash, count, right parentheses, right parentheses, semicolon. Line 40, indented twice: right brace. Line 41, indented twice: else. Line 42, indented thrice: System dot out dot print l n, left parentheses, double quotes, backslash n, No grades were entered, double quotes, right parentheses, semicolon. Line 43, indented once: right brace. Line 44: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, throws, int, new, if, else, while. Line 1: Forward slash. Asterisk. Find the maximum test grade. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: import java dot i o dot asterisk, semicolon. Line 7: blank. Line 8: public class Find Maximum Grade. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses, throws I O Exception. Line 11, indented once: left brace. Line 12, indented twice: int max Grade, semicolon. Line 13, indented twice: int grade, semicolon. Line 14: blank. Line 15, indented twice: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, grades dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 16: blank. Line 17, indented twice: System dot out dot print l n, left parentheses, double quotes, This program finds the maximum grade, blank space, double quotes. Line 18, indented multiple times: plus, double quotes, for a class, double quotes, right parentheses, semicolon. Line 19: blank. Line 20, indented twice: if, left parentheses, exclamation mark, file dot has Next, left parentheses, right parentheses, right parentheses. Line 21, indented twice: left brace. Line 22, indented thrice: System dot out dot print l n, left parentheses, double quotes, No test grades are in the file, double quotes, right parenthesese, semicoon. Line 23, indented twice: right brace. Line 24, indented twice: else. Line 25, indented twice: left brace. Line 26, indented thrice: max Grade, equals, file dot next Int, left parentheses, right parentheses, semicolon. 2 forward slashes, make first grade the max. Line 27: blank. Line 28, indented thrice: while, left parentheses, file dot has Next, left parentheses, right parentheses, right parentheses. Line 29, indented thrice: left brace. Line 30, indented 4 times: grade, equals, file dot next Int, left parentheses, right parentheses, semicolon. 2 forward slashes, read next grade. Line 31: blank. Line 32, indented 4 times: if, left parentheses, grade greater than max Grade, right parentheses. Line 33, indented 5 times: max Grade, equals, grade, semicolon. 2 forward slashes, save as current max. Line 34, indented thrice: right brace. Line 35: blank. Line 36, indented thrice: System dot out dot print l n, left parentheses, double quotes, The maximum grade is, blank space, double quotes, plus, max Grade, right parentheses, semicolon. Line 37, indented twice: right brace. Line 38, indented once: right brace. Line 39: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, new, int. Line 1: Forward slash. Asterisk. Reading an integer from the user. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: import java dot util dot Scanner, semicolon. Line 5: blank. Line 6: public class Read Integer. Line 7: left brace. Line 8, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: System dot out dot print, left parentheses, double quotes, Enter your age as an integer, right angled bracket, double quotes, right parentheses, semicolon. Line 13, indented twice: int age, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: System dot out dot print l n, left parentheses, double quotes, Your age is, blank space, double quotes, plus, age, right parentheses, semicolon. Line 16, indented once: right brace. Line 17: right brace.

Back to Example

The title of the table is, Selected Input Stream Testing Methods of the Scanner Class. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: boolean. Method name and argument list: has Next, left parentheses, right parentheses, returns true if there is another token in the input stream; false, otherwise. Row 2. Return value: boolean. Method name and argument list: has Next Byte, left parentheses, right parentheses, returns true if the token in the input stream can be read as a byte; false, otherwise. Row 3. Return value: boolean. Method name and argument list: has Next Short, left parentheses, right parentheses, returns true if the token in the input stream can be read as a short; false, otherwise. Row 4. Return value: boolean. Method name and argument list: has Next Int, left parentheses, right parentheses, returns true if the token in the input stream can be read as an int; false, otherwise. Row 5. Return value: boolean. Method name and argument list: has Next Long, left parentheses, right parentheses, returns true if the token in the input stream can be read as a long; false, otherwise. Row 6. Return value: boolean. Method name and argument list: has Next Float, left parentheses, right parentheses, returns true if the token in the input stream can be read as a float; false, otherwise. Row 7. Return value: boolean. Method name and argument list: has Next Double, left parentheses, right parentheses, returns true if the token in the input stream can be read as a double; false, otherwise. Row 8. Return value: boolean. Method name and argument list: has Next Boolean, left parentheses, right parentheses, returns true if the token in the input stream can be read as a boolean; false, otherwise. Row 9. Return value: String. Method name and argument list: next Line, left parentheses, right parentheses, returns the remainder of the input line as a String.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, String, new, while, int. Line 1: Forward slash. Asterisk. Type hyphen Safe Input

Using Scanner. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Type Safe Read Integer. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 12, indented twice: String garbage, semicolon. Line 13: blank. Line 14, indented twice: System dot out dot print, left parentheses, double quotes, Enter your age as an integer, right angled bracket, double quotes, right parentheses, semicolon. Line 15, indented twice: while, left parentheses, exclamation mark, scan dot has Next Int, left parentheses, right parentheses, right parentheses. Line 16, indented twice: left brace. Line 17, indented thrice: garbage, equals, scan dot next Line, left parentheses, right parentheses, semicolon. Line 18, indented thrice: System dot out dot print, left parentheses, double quotes, backslash, n, Please enter an integer, right angled bracket, double quotes, right parentheses, semicolon. Line 19, indented twice: right brace. Line 20, indented twice: int age, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 21, indented twice: System dot out dot print l n, left parentheses, double quotes, Your age is, double quotes, plus, age, right parentheses, semicolon. Line 22, indented once: right brace. Line 23: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, new, char, double, while, switch, case, default. Line 1: Forward slash. Asterisk. Order from a menu. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: import java dot text dot Decimal Format, semicolon. Line 7: blank. Line 8: public class Bonnies Burgers. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: String menu, equals, double quotes, backslash, t, uppercase c, Cheeseburger, colon, dollar sign, 7.49, double quotes, semicolon. Line 13, indented multiple times: menu, plus, equals, double quotes, backslash, n, backslash, t, uppercase h, Hot dog, colon, dollar sign, 6.99, double quotes, semicolon. Line 14, indented multiple times: menu, plus, equals, double quotes, backslash, n, backslash, t, uppercase l, Lemonade, colon, dollar sign, 2.50, double quotes, semicolon. Line 15, indented multiple times: menu, plus, equals, double quotes, backslash, n, backslash, t, uppercase t, Iced tea, colon, dollar sign, 2.75, double quotes, semicolon. Line 16: blank. Line 17, indented twice: Decimal Format money, equals, new Decimal

Format, left parentheses, double quotes, dollar sign, hash, decimal point, 00, double quotes, right parentheses, semicolon. Line 18, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 19: blank. Line 20, indented twice: System dot out dot print l n, left parentheses, double quotes, Welcome to Bonnie's Burgers, period, double quotes. Line 21, indented multiple times: plus, double quotes, Select from our menu, colon, blank space, double quotes, right parentheses, semicolon. Line 22: blank. Line 23, indented twice: System dot out dot print l n, left parentheses, menu, right parentheses, semicolon. 2 forward slashes, print the menu. Line 24, indented twice: System dot out dot print, left parentheses, double quotes, Order an item, comma, blank space, double quotes. Line 25, indented multiple times: plus, double quotes, or, blank space, backslash, double quotes, uppercase s, backslash, double quotes, to stop, right angled bracket, double quotes, right parentheses, semicolon. Line 26, indented twice: char option, equals, scan dot next, left parentheses, right parentheses, dot, char At, left parentheses, 0, right parentheses, semicolon. Line 27, indented twice: double order Cost, equals, 0.0, semicolon. Line 28: blank. Line 29, indented twice: while, left parentheses, option, exclamation mark, equals, single quote, uppercase s, single quote, 2 ampersand signs, option, exclamation mark, equals, single quote, lowercase s, single quote, right parentheses. Line 30, indented twice: left brace. Line 31, indented thrice: switch, left parentheses, option, right parentheses. Line 32, indented thrice: left brace. Line 33, indented 4 times: case, single quote, lowercase c, single quote, comma, single quote, uppercase c, single quote, hyphen, right angled bracket. Line 34, indented 4 times: left brace. Line 35, indented 5 times: System dot out dot print, left parentheses, double quotes, Cheeseburger ordered, period, double quotes, right parentheses, semicolon. Line 36, indented 5 times: order Cost, plus, equals, 7.49, semicolon. Line 37, indented 4 times: right brace. Line 38, indented 4 times: case, single quote, lowercase h, single quote, comma, single quote, uppercase h, single quote, hyphen, right angled bracket. Line 39, indented 4 times: left brace. Line 40, indented 5 times: System dot out dot print, left parentheses, double quotes, Hot dog ordered, period, double quotes, right parentheses, semicolon. Line 41, indented 5 times: order Cost, plus, equals, 6.99, semicolon. Line 42, indented 4 times: right brace. Line 43, indented 4 times: case, single quote, lowercase l, single quote, comma, single quote, uppercase l, single quote, hyphen, right angled bracket. Line 44, indented 4 times: left brace. Line 45, indented 5 times: System dot out dot print, left parentheses, double quotes, Lemonade ordered, period, blank space, double quotes, right parentheses, semicolon. Line 46, indented 5 times: order Cost, plus, equals, 2.50, semicolon. Line 47, indented 4 times: right brace. Line 48, indented 4 times: case, single quote, lowercase t, single quote, comma, single quote, uppercase t, single quote, hyphen, right angled bracket.

Line 49, indented 4 times: left brace. Line 50, indented 5 times: System dot out dot print, left parentheses, double quotes, Iced tea ordered, period, double quotes, right parentheses, semicolon. Line 51, indented 5 times: order Cost, plus, equals, 2.75, semicolon. Line 52, indented 4 times: right brace. Line 53, indented 4 times: default, hyphen, right angled bracket. Line 54, indented 5 times: System dot out dot print l n, left parentheses, double quotes, Unrecognized menu item, period, double quotes, right parentheses, semicolon. Line 55, indented 3 times: right brace. Line 56: blank. Line 57, indented 3 times: System dot out dot print l n, left parentheses, double quotes, Subtotal, colon, blank space, double quotes. Line 58, indented multiple times: plus, money dot format, left parentheses, order Cost, right parentheses, right parentheses, semicolon. Line 59: blank. Line 60, indented 3 times: System dot out dot print, left parentheses, double quotes, backslash, n, Order another item, comma, blank space, double quotes. Line 61, indented multiple times: plus, double quotes, or, backslash, double quotes, uppercase s, backslash, double quotes, to stop, right angled bracket, double quotes, right parentheses, semicolon. Line 62, indented 3 times: option, equals, scan dot next, left parentheses, right parentheses, dot, char At, left parentheses, 0, right parentheses, semicolon. Line 63, indented twice: right brace. Line 64: blank. Line 65, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, Total order cost is, double quotes. Line 66, indented multiple times:

plus, money, dot, format, left parentheses, order Cost, right parentheses, right parentheses, semicolon. Line 67, indented once: right brace. Line 68: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, int, new, do, while. Line 1: Forward slash. Asterisk. Validate input is between 1 and 10. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Validate Input. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: int number, semicolon. 2 forward slashes, input value. Line 12, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 13: blank. Line 14, indented twice: do. Line 15, indented twice: left brace. Line 16, indented thrice: System dot out dot print, left parentheses, double quotes, Enter a number between 1 and 10, right angled bracket, double quotes, right parentheses, semicolon. Line 17, indented thrice: number, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 18, indented twice: right brace, while, left parentheses, number less than 1, 2 pipes, number greater than 10, right parentheses, semicolon. Line 19: blank.

Line 20, indented twice: System dot out dot print l n, left parentheses, double quotes, Thank you, exclamation mark, double quotes, right parentheses, semicolon. Line 21, indented once: right brace. Line 22: right brace.

Back to Example

Program code. The code contains the keywords: public, void. Line 1: public void checkout, left parentheses, right parentheses. Line 2: left brace. Line 3, indented once: Forward slash. Asterisk. 5 asterisks. Student writes the body of this method. 5 asterisks. Asterisk. Forward slash. Line 4, indented once: 2 forward slashes. Line 5, indented once: 2 forward slashes. Using a while loop, comma, calculate the total price. Line 6, indented once: 2 forward slashes. of the groceries. Line 7, indented once: 2 forward slashes. Line 8, indented once: 2 forward slashes. The get Next method, left parentheses, in this Cashier Drawing class, right parentheses, returns the. Line 9, indented once: 2 forward slashes. next item on the conveyor belt, comma, which is an Item object. Line 10, indented once: 2 forward slashes. left parentheses, we do not know which item and we do not know how many items. Line 11, indented once: 2 forward slashes, are in the cart hyphen this is randomly generated, right parentheses. Line 12, indented once: 2 forward slashes, get Next does not take any arguments. Its A P I is, colon. Line 13, indented once: 2 forward slashes. Tab spaces. Item get Next, left parentheses, right parentheses. Line 14, indented once: 2 forward slashes. Line 15, indented once: 2 forward slashes. Right after you update the current subtotal. Line 16, indented once: 2 forward slashes, you should call the animate method. Line 17, indented once: 2 forward slashes, The animate method takes one argument, colon, a double, comma. Line 18, indented once: 2 forward slashes, which is your current subtotal. Line 19, indented once: 2 forward slashes. For example, if the name of your variable representing. Line 20, indented once: 2 forward slashes, the current subtotal is total, comma, your call to the animate. Line 21, indented once: 2 forward slashes, method should be, colon. Line 22, indented once: 2 forward slashes, animate, left parentheses, total, right parentheses, semicolon. Line 23, indented once: 2 forward slashes. Line 24, indented once: 2 forward slashes. The instance method get Price of the Item class. Line 25, indented once: 2 forward slashes. returns the price of the Item object. Line 26, indented once: 2 forward slashes. The method get Price does not take any arguments. Line 27, indented once: 2 forward slashes. Its A P I is, colon. Line 28, indented once: 2 forward slashes. Tab spaces. double get Price, left parentheses, right parentheses. Line 29, indented once: 2 forward slashes. Line 30, indented once: 2 forward slashes. The cart is empty when the get Next method returns. Line 31, indented once: 2 forward slashes. the divider item. Line 32, indented once: 2 forward slashes. You detect the divider item because its price. Line 33, indented once: 2 forward slashes. is negative 0.99. So an Item with a price of negative 0.99.

Line 34, indented once: 2 forward slashes. is the sentinel value for the loop. Line 35, indented once: 2 forward slashes. Line 36, indented once: 2 forward slashes. After you scan the divider, comma, print the total to the console. Line 37: blank. Line 38, indented once: 2 forward slashes. End of student code. Line 39: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, int, new, for. Line 1: Forward slash. Asterisk. Find the total of 5 numbers. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Sum 5 Numbers. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: int total, equals, 0, semicolon. 2 forward slashes, stores the sum of the 5 numbers. Line 12, indented twice: int number, semicolon. 2 forward slashes, stores the current input. Line 13: blank. Line 14, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 15: blank. Line 16, indented twice: for, left parentheses, int i equals 1, semicolon, i less than or equal to 5, semicolon, i plus, plus, right parentheses. Line 17, indented twice: left brace. Line 18, indented thrice: System dot out dot print, left parentheses, double quotes, Enter an integer, right angled bracket, double quotes, right parentheses, semicolon. Line 19, indented thrice: number, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 20: blank. Line 21, indented thrice: total, plus, equals, number, semicolon. 2 forward slashes, add input to total. Line 22, indented twice: right brace. Line 23: blank. Line 24, indented twice: 2 forward slashes. process results by printing the total. Line 25, indented twice: System dot out dot print l n, left parentheses, double quotes, The total is, double quotes, plus, total, right parentheses, semicolon. Line 26, indented once: right brace. Line 27: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, for, int. Line 1: Forward slash. Asterisk. Print the even numbers from zero to twenty. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Print Even. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: String to Print, equals, double quotes, double quotes, semicolon. 2 forward slashes, initialize output String. Line 10: blank. Line 11, indented twice: for, left parentheses, int i equals 0, semicolon, i less than or equal to 20, semicolon, i plus equals 2, right parentheses. Line 12, indented twice: left brace. Line 13, indented thrice: to Print, plus, equals, i, plus, double quotes, blank space, double quotes, semicolon. 2 forward slashes, append current number and a space. Line 14, indented twice: right brace. Line 15: blank. Line 16, indented twice: System dot out dot print l n, left parentheses, to Print, right parentheses, semicolon, 2 forward slashes, print results. Line 17, indented once: right brace. Line 18: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, new, if, else, boolean, int, for, char, true, false. Line 1: Forward slash. Asterisk. Count tokens in a sentence. Line 2: Asterisk. Tab space. Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: public class Count Tokens. Line 7: left brace. Line 8, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: System dot out dot print l n, left parentheses, double quotes, Enter a sentence, colon, double quotes, right parentheses, semicolon. Line 13, indented twice: String sentence, equals, scan dot next Line, left parentheses, right parentheses, semicolon. Line 14, indented twice: sentence, equals, sentence dot trim, left parentheses, right parentheses, semicolon. 2 forward slashes. remove leading, forward slash, trailing white space. Line 15: blank. Line 16, indented twice: if, left parentheses, sentence dot length, left parentheses, right parentheses, 2 equals sign, 0, right parentheses. Line 17, indented twice: left brace. Line 18, indented thrice: System dot out dot print l n, left parentheses, double quotes, The sentence is empty. double quotes, right parentheses, semicolon. Line 19, indented twice: right brace. Line 20, indented twice: else. Line 21, indented twice: left brace. Line 22, indented thrice: boolean in White Space, equals, false, semicolon. Line 23, indented thrice: int count Tokens, equals, 1, semicolon. Line 24: blank. Line 25, indented thrice: for, left parentheses, int i equals 0, semicolon, i less than sentence dot length, left parentheses, right parentheses, semicolon, i plus, plus, right parentheses. Line 26, indented thrice: left brace. Line 27, indented 4 times: char c, equals, sentence dot char At, left parentheses, i, right parentheses, semicolon. Line 28, indented 4 times: if, left parentheses, c, 2 equals sign, single quote, blank space, single quote, 2 pipes, c, 2 equals sign, single quote, backslash, t, single quote, right parentheses. Line 29, indented 4 times: left brace. Line 30, indented 5 times: if, left parentheses, exclamation mark, in White Space, right parentheses. Line 31, indented 5 times: left brace. Line 32, indented 6 times: count Tokens, plus, plus, semicolon. Line 33, indented 6 times: in White Space, equals, true, semicolon. Line 34, indented 5 times: right brace. Line 35, indented 4 times:

right brace. Line 36, indented 4 times: else. 2 forward slashes, not a white space character. Line 37, indented 4 times: left brace. Line 38, indented 5 times: in White Space, equals, false, semicolon. Line 39, indented 4 times: right brace. Line 40: blank. Line 41, indented 3 times: right brace. Line 42, indented 3 times: System dot out dot print I n, left parentheses, double quotes, The sentence contains, blank space, double quotes, plus, count Tokens. Line 43, indented multiple times: plus, left parentheses, count Tokens, 2 equals sign, 1, question mark, double quotes, token. double quotes, colon, double quotes, tokens. double quotes, right parentheses, right parentheses, semicolon. Line 44, indented twice: right brace. Line 45, indented once: right brace. Line 46: right brace.

Back to Example

The title of the table is, The trim Method of the String class. The table has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Row 1. Return value: String. Method name and argument list: trim, left parentheses, right parentheses, returns a String with leading and trailing white space characters removed.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, String, new, for, int. Line

1: Forward slash. Asterisk. Print a sentence backwards. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: import java dot util dot Scanner, semicolon. Line 5: blank. Line 6: public class Backwards. Line 7: left brace. Line 8, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: String backwards, equals, 2 double quotes, semicolon. Line 13: blank. Line 14, indented twice: System dot out dot print l n, left parentheses, double quotes, Enter a sentence, colon, double quotes, right parentheses, semicolon. Line 15, indented twice: String original, equals, scan dot next Line, left parentheses, right parentheses, semicolon. Line 16: blank. Line 17, indented twice: for, left parentheses, int i equals original dot length, left parentheses, right parentheses, minus, 1, semicolon, i greater than or equal to 0, semicolon, i minus, minus, right parentheses. Line 18, indented twice: left brace. Line 19, indented thrice: backwards, plus, equals, original dot char At, left parentheses, i, right parentheses, semicolon. Line 20, indented twice: right brace. Line 21: blank. Line 22, indented twice: System dot out dot print l n, left parentheses, double quotes, The sentence backwards is, colon, backslash, n, double quotes, plus, backwards, right parentheses, semicolon. Line 23, indented once: right

brace. Line 24: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, void, final, int, for, static. Line 1: Forward slash. Asterisk. Bull's hyphen eye Target. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java f x dot application dot Application, semicolon. Line 6: import java f x dot scene dot canvas dot Canvas, semicolon. Line 7: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 8: import java f x dot scene dot pain dot Color, semicolon. Line 9: import java f x dot stage dot Stage, semicolon. Line 10: blank. Line 11: public class Bulls Eye extends Application. Line 12: left brace. Line 13: blank. Line 14, indented once: at symbol, Override. Line 15, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 16, indented once: left brace. Line 17, indented twice: 2 forward slashes. set up window title and size. Line 18, indented twice: Graphics Context, g c, equals, J I Graphics Utility dot set Up Graphics, left parentheses. Line 19, indented multiple times: stage, comma, double quotes, Bull's Eye, double quotes, comma, 700, comma, 400, right parentheses, semicolon. Line 20: blank. Line 21, indented twice: 2 forward slashes. center of the bull's eye. Line 22, indented twice: final int CENTER underscore X equals 350, comma, CENTER underscore Y equals 200, semicolon. Line 23: blank. Line 24, indented twice: 2 forward slashes, start and end diameter, comma, amount to decrease diameter. Line 25, indented twice: final int START underscore DIAMETER equals 300, comma, END underscore DIAMETER equals 25, comma. Line 26, indented multiple times: DECREMENT equals 25, semicolon. Line 27: blank. Line 28, indented twice: 2 forward slashes, color of first circle. Line 29, indented twice: Color toggle Color, equals, Color dot BLACK, semicolon. Line 30, indented twice: g c dot set Fill, left parentheses, toggle Color, right parentheses, semicolon. Line 31: blank. Line 32, indented twice: for, left parentheses, int diameter equals START underscore DIAMETER, semicolon, diameter greater than or equal to END underscore DIAMETER, semicolon. Line 33, indented multiple times: diameter minus equals DECREMENT, right parentheses. Line 34, indented twice: left brace. Line 35, indented thrice: 2 forward slashes, draw circle with current diameter value. Line 36, indented thrice: g c dot fill Oval, left parentheses, CENTER underscore X minus diameter forward slash 2, comma, CENTER underscore Y minus diameter forward slash 2, comma. Line 37, indented multiple times: diameter comma diameter, right parentheses, semicolon. Line 38, indented thrice: 2 forward slashes, switch color. Line 39, indented thrice: if, left parentheses, toggle Color dot equals, left parentheses, Color dot BLACK, right parentheses, right parentheses. Line 40, indented 4 times: toggle Color, equals, Color dot RED, semicolon. Line 41, indented thrice: else. Line 42, indented 4 times:

toggle Color, equals, Color dot BLACK, semicolon. Line 43: blank. Line 44, indented thrice: g c dot set Fill, left parentheses, toggle Color, right parentheses, semicolon. Line 45: blank. Line 46, indented twice: right brace. Line 47, indented once: right brace. Line 49: blank. Line 49, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 50, indented once: left brace. Line 51, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 52, indented once: right brace. Line 53: right brace.

Back to Example

Line 18: blank. Line 19, indented thrice: System dot out dot print l n, left parentheses, right parentheses, semicolon. 2 forward slashes. print a newline. Line 20, indented twice: right brace. Line 21, indented once: right brace. Line 22: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, int, final, boolean, new, do, while, false, for, if, true. Line 1: Forward slash. Asterisk. Factors of integers. Line 2, indented once: with checks for primes. Line 3, indented once: Anderson, comma, Franceschi. Line 4: Asterisk. Forward slash. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Factors. Line 8: left brace. Line

9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented once: int number, semicolon. 2 forward slashes, positive integer entered by user. Line 12, indented once: final int SENTINEL equals 0, semicolon. Line 13, indented once: boolean factors Found, semicolon. 2 forward slashes, flag signals whether factors are found. Line 14: blank. Line 15, indented once: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 16: blank. Line 17, indented once: 2 forward slashes, priming read. Line 18, indented once: do. Line 19, indented once: left brace. Line 20, indented twice: System dot out dot print, left parentheses, double quotes, Enter a positive integer, blank space, double quotes. Line 21, indented multiple times: plus, double quotes, or 0 to exit, right angled bracket, blank space, double quotes, right parentheses, semicolon. Line 22, indented twice: number, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 23, indented once: right brace, while, left parentheses, number less than 0, right parentheses, semicolon. Line 24: blank. Line 25, indented once: while, left parentheses, number, exclamation mark, equals, SENTINEL, right parentheses. Line 26, indented once: left brace. Line 27, indented twice: System dot out dot print, left parentheses, double quotes, Factors of, double quotes, plus, number, plus, double quotes, colon, blank space, double quotes, right parentheses, semicolon. Line 28, indented twice: factors Found, equals, false, semicolon. 2 forward slashes, reset flag to no factors. Line 29: blank. Line 30, indented twice: for, left parentheses, int factor, equals, 2, semicolon, factor less than or equal to number forward slash 2, semicolon, factor, plus, plus, right parentheses. Line 31, indented twice: left brace. Line 32, indented thrice: if, left parentheses, number, percentage sign, factor, 2 equals sign, 0, right parentheses. Line 33, indented thrice: left brace. Line 34, indented 4 times: System dot out dot print, left parentheses, factor, plus, double quotes, blank space, double quotes, right parentheses, semicolon. Line 35, indented 4 times: factors Found, equals, true, semicolon. Line 36, indented thrice: right brace. Line 37, indented twice: right brace. 2 forward slashes, end of for loop. Line 38: blank. Line 39, indented thrice: if, left parentheses, exclamation mark, factors Found, right parentheses. Line 40, indented 4 times: System dot out dot print, left parentheses, double quotes, none, comma, blank space, double quotes, plus, number, plus, double quotes, is prime, double quotes, right parentheses, semicolon. Line 41: blank. Line 42, indented thrice: System dot out dot print l n, left parentheses, right parentheses, semicolon, 2 forward slashes, print a newline. Line 43, indented thrice: System dot out dot print l n, left parentheses, right parentheses, semicolon. 2 forward slashes, skip a line. Line 44: blank. Line 45, indented thrice: 2 forward slashes, read next number. Line 46, indented thrice: do. Line 47, indented thrice: left brace. Line 48, indented 4 times: System dot out dot print, left parentheses, double quotes, Enter a positive integer, double quotes. Line 49, indented multiple times: plus, double quotes, or 0 to exit, right angled bracket, double quotes, right parentheses, semicolon. Line 50, indented 4 times: number, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 51, indented thrice: right brace, while, left parentheses, number less than 0, right parentheses, semicolon. Line 52, indented twice: right brace. 2 forward slashes, end of while loop. Line 53, indented once: right brace. Line 54: right brace.

Back to Example

Program code. The code contains the following keywords: public, void. Line 1: public void checkout, left parentheses, int number Of Items, right parentheses. Line 2: left brace. Line 3, indented once: Forward slash. Asterisk. 5 asterisks. Student writes the body of this method. 5 asterisks. Asterisk. Forward slash. Line 4, indented once: 2 forward slashes. Line 5, indented once: 2 forward slashes. The parameter of this method, comma, number Of Items, comma. Line 6, indented once: 2 forward slashes. represents the number of items in the cart. The. Line 7, indented once: 2 forward slashes, user will be prompted for this number. Line 8, indented once: 2 forward slashes. Line 9, indented once: 2 forward slashes, Using a for loop, comma, calculate the total price. Line 10, indented once: 2 forward slashes, of the groceries in the cart. Line 11, indented once: 2 forward slashes. Line 12, indented once: 2 forward slashes, The get Next method, left parentheses, in this Cashier Drawing class, right parentheses, returns the next. Line 13, indented once: 2 forward slashes, item in the cart, comma, which is an Item object, left parentheses, we do not. Line 14, indented once: 2 forward slashes, know which item will be returned, semicolon, this is randomly generated, right parentheses. Line 15, indented once: 2 forward slashes, get Next does not take any arguments. Its A P I is. Line 16, indented once: 2 forward salshes, Tab space. Item get Next, left parentheses, right parentheses. Line 17, indented once: 2 forward slashes. Line 18, indented once: 2 forward slashes. As the last statement of the body of your for loop, comma. Line 19, indented once: 2 forward slashes, you should call the animate method. Line 20, indented once: 2 forward slashes, The animate method takes one argument a double, comma. Line 21, indented once: 2 forward slashes, which is your current subtotal. Line 22, indented once: 2 forward slashes, For example, comma, if the name of your variable representing. Line 23, indented once: 2 forward slashes. the current subtotal is total, comma, your call to the animate. Line 24, indented once: 2 forward slashes. method should be, colon. Line 25, indented once: 2 forward slashes. Tab space. animate, left parentheses, total, right parentheses, semicolon. Line 26, indented once: 2 forward slashes. Line 27, indented once: 2 forward slashes, The get Price method of the Item class. Line 28, indented once: 2 forward slashes, returns the price of the Item object as a double. LIne 29, indented once: 2 forward slashes, The get Price method does not take any arguments. Its A P I is. Line 30, indented once: 2 forward slashes, double get Price, left parentheses, right parentheses. Line 31, indented once: 2 forward slashes. Line 32, indented once: 2 forward slashes, End of student code. Line 33, indented once: 2 forward slashes. Line 34, indented once: 2 forward slashes.

Back to Example

The table has 2 columns labeled Access Modifier and Class or Member can Be Referenced by, dot, dot, dot. The row entries are as follows. Row 1. Access Modifier: public. Class or Member can Be Referenced by, dot, dot, dot: methods of the same class, as well as methods of other classes. Row 2. Access Modifier: private. Class or Member can Be Referenced by, dot, dot, dot: methods of the same class only. Row 3. Access Modifier: protected. Class or Member can Be Referenced by, dot, dot, dot: methods in the same class, as well as methods of subclasses and methods in classes in the same package. Row 4. Access Modifier: no modifier, left parentheses, package access, right parentheses. Class or Member can Be Referenced by, dot, dot, dot: methods in the same package or same folder only.

Back to Table

The table has 2 columns labeled Data Type and Initial

Value. The row entries are as follows. Row 1. Data Type: byte. Initial Value: 0. Row 2. Data Type: short. Initial Value: 0. Row 3. Data Type: int. Initial Value: 0. Row 4. Data Type: long. Initial Value: 0. Row 5. Data Type: float. Initial Value: 0.0. Row 6. Data Type: double. Initial Value: 0.0. Row 7. Data Type: char. Initial Value: null character, left parentheses, single quote, back slash, u, 0 0 0 0, single quote, right parentheses. Row 8. Data Type: boolean. Initial Value: false. Row 9. Data Type: object reference. Initial Value: null.

Back to Table

Program code. The code contains the following keywords: public, class, private, String, int, double, if. Line 1: Forward slash. Asterisk. Auto class, comma, Version 1. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Auto. Line 6: left brace. Line 7, indented once: 2 forward slashes. instance variables. Line 8, indented once: private String model, semicolon. 2 forward slashes, model of auto. Line 9, indented once: private int miles Driven, semicolon. 2 forward slashes. number of miles driven. Line 10, indented once: private double gallons Of Gas, semicolon. 2 forward slashes. number of gallons of gas. Line 11: blank. Line 12, indented once: 2 forward slashes. Default constructor, colon. Line 13, indented once: 2 forward slashes. initializes model to, double quotes, unknown, double quotes, semicolon. Line 14, indented once: 2 forward slashes. miles Driven is auto initialized to 0. Line 15, indented once: 2 forward slashes. Tab space. and gallons Of Gas to 0.0. Line 16, indented once: public Auto, left parentheses, right parentheses. Line 17, indented once: left brace. Line 18, indented twice: model, equals, double quotes, unknown, double quotes, semicolon. Line 19, indented once: right brace. Line 20: blank. Line 21, indented once: 2 forward slashes. Overloaded constructor, colon. Line 22, indented once: 2 forward slashes. allows client to set begining values for. Line 23, indented once: 2 forward slashes. model, comma, miles Driven, comma, and gallons Of Gas. Line 24, indented once: public Auto, left parentheses, String start Model, comma. Line 25, indented multiple times: int start Miles Driven, comma. Line 26, indented multiple times: double start Gallons Of Gas, right parentheses. Line 27, indented once: left brace. Line 28, indented twice: model, equals, start Model, semicolon. Line 29: blank. Line 30, indented twice: 2 forward slashes. validate start Miles Driven parameter. Line 31, indented twice: if, left parentheses, start Miles Driven greater than or equal to 0, right parentheses. Line 32, indented thrice: miles Driven, equals, start Miles Driven, semicolon. Line 33: blank. Line 34, indented thrice: 2 forward slashes. validate start Gallons Of Gas parameter. Line 35, indented twice: if, left parentheses, start Gallons Of Gas greater than or equal to 0.0, right parentheses. Line 36, indented thrice: gallons Of Gas, equals, start Gallons Of Gas, semicolon. Line 37, indented once: right brace. Line 38: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, new. Line 1: Forward slash. Asterisk. Auto Client, comma, Version 1. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Auto Client. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: System dot out dot print l n, left parentheses, double quotes, Instantiate sedan, double quotes, right parentheses, semicolon. Line 10, indented twice: Auto sedan, equals, new Auto, left parentheses, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash n, Instantiate s u v, double quotes, right parentheses, semicolon. Line 13, indented twice: Auto s u v, equals, new Auto, left parentheses, double qutoes, Trailblazer, double quotes, comma, 7000, comma, 437.5, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, Instantiate mini, double quotes, right parentheses, semicolon. Line 16, indented twice: 2 forward slashes. attempt to set invalid value for gallons of gas. Line 17, indented twice: Auto mini, equals, new Auto, left parentheses, double quotes, Mini Cooper, double quotes, comma, 200, comma, negative 1.0, right parentheses, semicolon. Line 18, indented once: right brace. Line 19: right brace.

Back to Example

A method in a class can access, the instance variables of its class, any parameters sent to the method, any variable the method declares within its body from the point of declaration until the end of the method or until the end of the block in which the variable was declared, whichever comes first, any methods in the class.

Back to Table

Program code. The code contains the following keywords: public, class, private, String, int, double, if, return. Line 1: Forward slash. Asterisk. Auto class, comma, Version 2. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Auto. Line 6: left brace. Line 7, indented once: 2 forward slashes, instance variables. Line 8, indented once: private String model, semicolon. 2 forward slashes, model of auto. Line 9, indented once: private int miles Driven, semicolon. 2 forward slashes, number of miles driven. Line 10, indented once: private double gallons Of Gas, semicolon. 2 forward slashes, number of gallons of gas. Line 11: blank. Line 12, indented once: 2 forward slashes. Default constructor, colon. Line 13, indented once: 2 forward slashes. initializes model to double quotes, unknown, double quotes, semicolon. Line 14, indented once: 2 forward slashes, miles Driven is autoinitizlized to 0. Line 15, indented once: 2 forward slashes. Tab space. and gallons Of Gas to 0.0. Line 16, indented once: public Auto, left parentheses, right parentheses. Line 17, indented once: left brace. Line 18, indented twice: model, equals, double quotes, unknown, double quotes, semicolon. Line 19, indented once: right brace. Line 20: blank. Line 21, indented once: 2 forward slashes. Overloaded constructor, colon. Line 22, indented once: 2 forward slashes, allows client to set beginning values for. Line 23, indented once: 2 forward slashes, model, comma, miles Driven, comma, and gallons Of Gas. Line 24, indented once: public Auto, left parentheses, String start Model, comma. Line 25, indented multiple times: int start Miles Driven, comma. Line 26, indented multiple times: double start Gallons Of Gas, right parentheses. Line 27, indented once: left brace. Line 28, indented twice: model, equals, start Model, semicolon. Line 29: blank. Line 30, indented twice: 2 forward slashes, validate start Miles Driven parameter. Line 31, indented twice: if, left parentheses, start Miles Driven greater than or equal to 0, right parentheses. Line 32, indented thrice: miles Driven, equals, start Miles Driven, semicolon. Line 33: blank. Line 34, indented twice: 2 forward salshes. validate start Gallons Of Gas parameter. Line 35, indented twice: if, left parentheses, start Gallons Of Gas greater than or equal to 0.0, right parentheses. Line 36, indented thrice: gallons Of Gas, equals, start Gallons Of Gas, semicolon. Line 37, indented once: right brace. Line 38: blank. Line 39, indented once: 2 forward slashes. Accessor method, colon. Line 40, indented once: 2 forward slashes, returns current value of model. Line 41, indented once: public String get Model, left parentheses, right parentheses. Line 42, indented once: left brace. Line 43, indented twice: return model, semicolon. Line 44, indented once: right brace. Line 45: blank. Line 46, indented once: 2 forward slashes. Accessor method, colon. Line 47, indented once: 2 forward slashes, returns current value of miles Driven. Line 48, indented once: public int get Miles Driven, left parentheses, right parentheses. Line 49, indented once: left brace. Line 50, indented once: return miles Driven, semicolon. Line 51, indented once: right brace. Line 52: blank. Line 53, indented once: 2 forward slashes, Accessor method, colon. Line 54, indented once: 2 forward slashes, returns current value of gallons Of Gas. Line 55, indented once: public double get Gallons Of Gas, left parentheses, right parentheses. Line 56, indented once: left brace. Line 56, indented twice: return gallons Of Gas, semicolon. Line 58, indented once: right brace. Line 59: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, new, int, double. Line 1: Forward slash. Asterisk. Auto Client, comma, Version 2. Line 2, indented once: Anderson, comma, Franceschi.

Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Auto Client. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: Auto sedan, equals, new Auto, left parentheses, right parentheses, semicolon. Line 10, indented twice: String sedan Model, equals, sedan dot get Model, left parentheses, right parentheses, semicolon. Line 11, indented twice: int sedan Miles, equals, sedan dot get Miles Driven, left parentheses, right parentheses, semicolon. Line 12, indented twice: double sedan Gallons, equals, sedan dot get Gallons Of Gas, left parentheses, right parentheses, semicolon. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, sedan, colon, model is, double quotes, plus, sedan Model. Line 14, indented multiple times: plus, double quotes, backslash, n, miles driven is, double quotes, plus, sedan Miles. Line 15, indented multiple times: plus, double quotes, backslash, n, gallons of gas is, double quotes, plus, sedan Gallons, right parentheses, semicolon. Line 16: blank. Line 17, indented twice: Auto s u v, equals, new Auto, left parentheses, double quotes, Trailblazer, double quotes, comma, 7000, comma, 437.5, right parentheses, semicolon. Line 18, indented twice: String s u v Model, equals, s u v dot get Model, left parentheses, right parentheses, semicolon. Line 19, indented twice: int s u v Miles, equals, s u v dot get Miles Driven, left parentheses, right parentheses, semicolon. Line 20, indented twice: double s u v Gallons, equals, s u v dot get Gallons Of Gas, left parentheses, right parentheses, semicolon. Line 21, indented tice: System dot out dot print l n, left parentheses, double quotes, s u v, colon, model is, blank space, double quotes, plus, s u v Model. Line 22, indented multiple times: plus, double quotes, backslash, n, miles driven is, blank space, double quotes, plus, s u v Miles. Line 23, indented multiple times: plus, double quotes, backslash, n, gallons of gas is, double quotes, plus, s u v Gallons, right parentheses, semicolon. Line 24, indented once: right brace. Line 25: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, private, String, int, double, return, if. Line 1: Forward slash. Asterisk. Auto class, comma, Version 3. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Auto. Line 6: left brace. Line 7, indented once: 2 forward slashes. instance variables. Line 8, indented once: private String model, semicolon. 2 forward slashes, model of auto. Line 9, indented once: private int miles Driven, semicolon. 2 forward slashes, number of miles driven. Line 10, indented once: private double gallons Of Gas, semicolon. 2 forward slashes, number of gallons of gas. Line 11: blank. Line 12, indented once: 2 forward slashes. Default constructor, colon. Line 13, indented once: 2 forward slashes. initializes model to, double quotes, unknown, double quotes, semicolon. Line 14, indented once: 2 forward slashes. miles Driven is autoinitialized to 0. Line 15, indented once: 2 forward slashes, and gallons Of Gas to 0.0. Line 16, indented once: public Auto, left parentheses, right parentheses. Line 17, indented once: left brace. Line 18, indented twice: model, equals, double quotes, unknown, double quotes, semicolon. Line 19, indented once: right brace. Line 20: blank. Line 21, indented once: 2 forward slashes. Overloaded constructor, colon. Line 22, indented once: 2 forward slashes. allows client to set beginning values for. Line 23, indented once: 2 forward slashes, model, comma, miles Driven, comma, and gallons Of Gas. Line 24, indented once: public Auto, left parentheses, String start Model, comma. Line 25, indented multiple times: int start Miles Driven, comma. Line 26, indented multiple times: double start Gallons Of Gas, right parentheses. Line 27, indented once: left brace. Line 28, indented twice: model, equals, start Model, semicolon. Line 29, indented twice: set Miles Driven, left parentheses, start Miles Driven, right parentheses, semicolon. Line 30, indented twice: set Gallons Of Gas, left parentheses, start Gallons Of Gas, right parentheses, semicolon. Line 31, indented once: right brace. Line 32: blank. Line 33, indented once: 2 forward slashes. Accessor method, colon. Line 34, indented once: 2 forward slashes. returns current value of model. Line 35, indented once: public String get Model, left parentheses, right parentheses. Line 36, indented once: left brace. Line 37, indented twice:

return model, semicolon. Line 38, indented once: right brace. Line 39: blank. Line 40, indented once: 2 forward slashes. Accessor method, colon. Line 41, indented once: 2 forward slashes. returns current value of miles Driven. Line 42, indented once: public int get Miles Driven, left parentheses, right parentheses. Line 43, indented once: left brace. Line 44, indented twice: return miles Driven, semicolon. Line 45, indented once: right brace. Line 46: blank. LIne 47, indented once: 2 forward slashes. Accessor method. Line 48, indented once: 2 forward slashes. returns current value of gallons Of Gas. Line 49, indented once: public double get Gallons Of Gas, left parentheses, right parentheses. Line 50, indented once: left brace. Line 51, indented twice: return gallons Of Gas, semicolon. Line 52, indented once: right brace. Line 53: blank. Line 54, indented once: 2 forward slashes. Mutator method, colon. Line 55, indented once: 2 forward slashes. allows client to set model. Line 56, indented once: public void set Model, left parentheses, String new Model, right parentheses. Line 57, indented once: left brace. Line 58, indented twice: model, equals, new Model, semicolon. Line 59, indented once: right brace. Line 60: blank. Line 61, indented once: 2 forward slashes. Mutator method, colon. Line 62, indented once: 2 forward slashes, allows client to set value of miles Driven, semicolon. Line 63, indented once: 2 forward slashes. if new value is not less than 0. Line 64, indented once: public void set Miles Driven, left parentheses, int new Miles Driven, right parentheses. Line 65, indented once: left brace. Line 66, indented twice: if, left parentheses, new Miles Driven, greater than or equal to, 0, right parentheses. Line 67, indented thrice: miles Driven, equals, new Miles Driven, semicolon. Line 68, indented once: right brace. Line 69: blank. Line 70, indented once: 2 forward slashes. Mutator method, colon. Line 71, indented once: 2 forward slashes, allows client to set value of gallons Of Gas, semicolon. Line 72, indented once: 2 forward slashes, if new value is not less than 0.0. Line 73, indented once: public void set Gallons Of Gas, left parentheses, double new Gallons Of Gas, right parentheses. Line 74, indented once: left brace. Line 75, indented twice: if, left parentheses, new Gallons Of Gas, greater than or equal to, 0.0, right parentheses. Line 76, indented thrice: gallons Of Gas, equals, new Gallons Of Gas, semicolon. Line 77, indented once: right brace. Line 78: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, new. Line 1: Forward slash. Asterisk. Auto Client, comma, Version 3. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Auto Client. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: Auto s u v, equals, new Auto, left parentheses, double quotes, Trailblazer, double quotes, comma, 7000, comma, 437.5, right parentheses, semicolon. Line 10: blank. Line 11, indented twice: 2 forward slashes, print initial values of instance variables. Line 12, indented twice: System dot out dot print l n, left parentheses, double quotes, s u v, colon, model is, blank space, double quotes, plus, s u v dot get Model, left parentheses, right parentheses. Line 13, indented multiple times: plus, double quotes, backslash, n, miles driven is, blank space, double quotes, plus, s u v dot get Miles Driven, left parentheses, right parentheses. Line 14, indented multiple times: plus, double quotes, backslash, n, gallons of gas is, blank space, double quotes, plus, s u v dot get Gallons Of Gas, left parentheses, right parentheses, right parentheses, semicolon. Line 15: blank. Line 16, indented twice: 2 forward slashes, call mutator method for each instance variable. Line 17, indented twice: s u v dot set Model, left parentheses, double quotes, Sportage, double quotes, right parentheses. Line 18, indented twice: s u v dot set Miles Driven, left parentheses, 200, right parentheses, semicolon. Line 19, indented twice: s u v dot set Gallons Of Gas, left parentheses, 10.5, right parentheses, semicolon. Line 20: blank. Line 21, indented twice: 2 forward slashes, print new values of instance variables. Line 22, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, s u v, colon, model is, blank space, double quotes, plus, s u v dot get Model, left parentheses, right parentheses. Line 23, indented multiple times: plus, double quotes, backslash, n, miles driven is, blank space, double quotes, plus, s u v dot get Miles Driven, left parentheses, right parentheses. Line 24, indented multiple times: plus, double quotes, backslash, n, gallons of gas is, blank space, double quotes, plus, s u v dot get Gallons Of Gas, left parentheses, right parentheses, right parentheses, semicolon. Line 25: blank. Line 26, indented twice: 2 forward slashes, attempt to set invalid value for miles Driven. Line 27, indented twice: s u v dot set Miles Driven, left parentheses, negative 1, right parentheses, semicolon. Line 28, indented twice: 2 forward slashes, print current values of instance variables. Line 29, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, s u v, colon, model is, blank space, double quotes, plus, s u v dot get Model, left parentheses, right parentheses. Line 30, indented multiple times: plus, double quotes, backslash, n, miles driven is, blank space, double quotes, plus, s u v dot get Miles Driven, left parentheses, right parentheses. Line 31, indented multiple times: plus, double quotes, backslash, n, gallons of gas is, double quotes, plus, s u v dot get Gallons Of Gas, left parentheses, right parentheses, right parentheses, semicolon. Line 32, indented once: right brace. Line 33: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, private, String, int, double, return, void, if, else. Line 1: Forward slash. Asterisk. Auto class, comma, Version 4. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Auto. Line 6: left brace. Line 7, indented once: 2 forward slashes, instance variables. Line 8, indented once: private String model, semicolon. 2 forward slashes, model of auto. Line 9, indented once: private int miles Driven, semicolon. 2 forward slashes, number of miles driven. Line 10, indented once: private double gallons Of Gas, semicolon. 2 forward slashes, number of gallons of gas. Line 11: blank. Line 12, indented once: 2 forward slashes. Default constructor, colon. Line 13, indented once: 2 forward slashes, initializes model to, double quotes, unknown, double quotes, semicolon. Line 14, indented once: 2 forward slashes, miles Driven is auto initialized to 0. Line 15, indented once: 2 forward slashes, and gallons Of Gas to 0.0. Line 16, indented once: public Auto, left parentheses, right parentheses. Line 17, indented once: left brace. Line 18, indented twice: model, equals, double quotes, unknown, double quotes, semicolon. Line 19, indented once: right brace. Line 20: blank. Line 21, indented once: 2 forward slashes, Overloaded constructor, colon. Line 22, indented once: 2 forward slashes, allows client to set beginning values for. Line 23, indented once: 2 forward slashes, model, comma, miles Driven, comma, and gallons Of Gas. Line 24, indented once: public Auto, left parentheses, String start Model, comma. Line 25, indented multiple times: int start Miles Driven, comma. Line 26, indented multiple times: double start Gallons Of

Gas, right parentheses. Line 27, indented once: left brace. Line 28, indented twice: model, equals, start Model, semicolon. Line 29, indented twice: set Miles Driven, left parentheses, start Miles Driven, right parentheses, semicolon. Line 30, indented twice: set Gallons Of Gas, left parentheses, start Gallons Of Gas, right parentheses, semicolon. Line 31, indented once: right brace. Line 32: blank. Line 33, indented once: 2 forward slashes, Accessor method, colon. Line 34, indented once: 2 forward slashes, returns current value of model. Line 35, indented once: public String get Model, left parentheses, right parentheses. Line 36, indented once: left brace. Line 37, indented twice: return model, semicolon. Line 38, indented once: right brace. Line 39: blank. Line 40, indented once: 2 forward slashes, Accessor method. Line 41, indented once: 2 forward slashes, returns current value of miles Driven. Line 42, indented once: public int get Miles Driven, left parentheses, right parentheses. Line 43, indented once: left brace. Line 44, indented twice: return miles Driven, semicolon. Line 45, indented once: right brace. Line 46: blank. Line 47, indented once: 2 forward slashes. Accessor method, colon. Line 48, indented once: 2 forward slashes, returns current value of gallons Of Gas. Line 49, indented once: public double get Gallons Of Gas, left parentheses, right parentheses. Line 50, indented once: left brace. Line 51, indented twice: return gallons Of Gas, semicolon. Line 52, indented once: right brace. Line 53: blank. Line 54, indented once: 2 forward slashes, Mutator method, colon. Line 55, indented once: 2 forward slashes, allows client to set model. Line 56, indented once: public void set Model, left parentheses, String new Model, right parentheses. Line 57, indented once: left brace. Line 58, indented twice: model, equals, new Model, semicolon. Line 59, indented once: right brace. Line 60: blank. Line 61, indented once: 2 forward slashes, Mutator method, colon. Line 62, indented once: 2 forward slashes, allows client to set value of miles Driven, semicolon. Line 63, indented once: 2 forward slashes, if new value is not less than 0. Line 64, indented once: public void set Miles Driven, left parentheses, int new Miles Driven, right parentheses. Line 65, indented once: left brace. Line 66, indented twice: if, left parentheses, new Miles Driven, greater than or equal to 0, right parentheses. Line 67, indented thrice: miles Driven, equals, new Miles driven, semicolon. Line 68, indented once: right brace. Line 69: blank. Line 70, indented once: 2 forward slashes, Mutator method, colon. Line 71, indented once: 2 forward slashes, allows client to set value of gallons Of Gas, semicolon. Line 72, indented once: 2 forward slashes, if new value is not less than 0.0. Line 73, indented once: public void set Gallons Of Gas, left parentheses, double new Gallons Of Gas, right parentheses. Line 74, indented once: left brace. Line 75, indented twice: if, left parentheses, new Gallons Of Gas, greater than or equal to, 0.0, right parentheses. Line 76, indented thrice: gallons Of Gas, equals, new Gallons Of Gas, semicolon. Line 77, indented once: right brace. Line 78: blank. Line 79, indented once: 2 forward slashes, Calculates miles per gallon. Line 80, indented once: 2 forward slashes. if no gallons of gas have been used, comma, returns 0.0, semicolon. Line 81, indented once: 2 forward slashes, otherwise, comma, returns miles per gallon. Line 82, indented once: 2 forward slashes, as miles Driven, forward slash, gallons Of Gas. Line 83, indented once: public double miles Per Gallon, left parentheses, right parentheses. Line 84, indented once: left brace. Line 85, indented twice: if, left parentheses, gallons Of Gas, greater than or equal to, 0.0001, right parentheses. Line 86, indented thrice: return miles Driven, forward slash, gallons Of Gas, semicolon. Line 87, indented twice: else. Line 88, indented thrice: return 0.0, semicolon. Line 89, indented once: right brace. Line 90: blank. Line 91, indented once: 2 forward slashes, Calculates money spent on gas. Line 92, indented once: 2 forward slashes, returns price per gallon times gallons of gas. Line 93, indented once: public double money Spent On Gas, left parentheses, double price Per Gallon, right parentheses. Line 94, indented once: left brace. Line 95, indented twice: return price Per Gallon, asterisk, gallons Of Gas, semicolon. Line 96, indented once: right brace. Line 97: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, new, double. Line 1: Forward slash. Asterisk. Auto Client, comma, Version 4. Line 2, indented once: Anderson, comma, Franceschi.

Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot text dot Decimal Format, semicolon. Line 6: import java dot text dot Number Format, semicolon. Line 7: blank. Line 8: public class Auto Client. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: Auto s u v, equals, new Auto, left parentheses, double quotes, Trailblazer, double quotes, comma, 7000, comma, 437.5, right parentheses, semicolon. Line 13: blank. Line 14, indented twice: double mileage, equals, s u v dot miles Per Gallon, left parentheses, right parentheses, semicolon. Line 15, indented twice: Decimal Format m p g Format, equals, new Decimal Format, left parentheses, double quotes, 0.0, double quotes, right parentheses, semicolon. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, Mileage for s u v is, blank space, double quotes. Line 17, indented multiple times: plus, m p g Format dot format, left parentheses, mileage, right parentheses, right parentheses, semicolon. Line 18: blank. Line 19, indented twice: double gas Cost, equals, s u v dot money Spent On Gas, left parentheses, 2.79, right parentheses, semicolon. Line 20, indented twice: Number Format money, equals, Number Format dot get Currency Instance, left parentheses, right parentheses, semicolon. Line 21, indented twice: System dot out dot print l n, left parentheses, double quotes, Gas cost for s u v is, blank space, doubel quotes. Line 22, indented multiple times: plus, money dot format, left parentheses, gas Cost, right parentheses, right parentheses, semicolon. Line 23, indented once: right brace. Line 24: right brace.

Back to Example

Line 1: Forward slash, asterisk. Airport class. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Airport. Line 6: Left brace. Line 7: 2 forward slashes. 1. 5 asterisks. Define the instance variables 5 asterisks. Line 8: 2 forward slashes. airport Code is a String. Line 9: 2 forward slashes. gates is an integer. Line 10: blank. Line 11: blank. Line 12: blank. Line 13: 2 forward slashes. 2. 5 asterisks. Write this method 5 asterisks. Line 14: 2 forward slashes. Default constructor: Line 15: 2 forward slashes. method name: Airport. Line 16: 2 forward slashes. return value: none. Line 17: 2 forward slashes. parameters: none. Line 18: 2 forward slashes. function: sets the airport Code to an empty String. Line 19: blank. Line 20: blank. Line 21: blank. Line 22: 2 forward slashes. 3. 5 asterisks. Write this method 5 asterisks. Line 23: 2 forward slashes. Overloaded constructor: Line 24: 2 forward slashes. method name: Airport. Line 25: 2 forward slashes. return value: none. Line 26: 2 forward slashes. parameters: a String start Airport Code and an int start Gates. Line 27: 2 forward slashes. function. Line 28: 2 forward slashes. calls the set Airport Code method. Line 29: 2 forward slashes. passing start Airport Code parameter. semicolon.

Line 30: 2 forward slashes. calls the set Gates method, passing start Gates parameter. Line 31: blank. Line 32: blank. Line 33: blank. Line 34: blank. Line 35: 2 forward slashes. 4. 5 asterisks. Write this method 5 asterisks. Line 36: 2 forward slashes. Accessor method for the airport Code instance variable. Line 37: 2 forward slashes. method name: get Airport Code. Line 38: 2 forward slashes. return value: String. Line 39: 2 forward slashes. parameters: none. Line 40: 2 forward slashes. function: returns airport Code. Line 41: blank. Line 42: blank. Line 43: blank. Line 44: 2 forward slashes. 5. 5 asterisks. Write this method 5 asterisks. Line 45: 2 forward slashes. Accessor method for the gates instance variable. Line 46: 2 forward slashes. method name: get Gates. Line 47: 2 forward slashes. return value: int. Line 48: 2 forward slashes. parameters: none. Line 49: 2 forward slashes. function: returns gates. Line 50: blank. Line 51: blank. Line 52: blank. Line 53: 2 forward slashes. 6. 5 asterisks. Write this method 5 asterisks. Line 54: 2 forward slashes. Mutator method for the airport Code instance variable. Line 55: 2 forward slashes. method name: set Airport Code. Line 56: 2 forward slashes. return value: void. Line 57: 2 forward slashes. parameters: String new Airport Code. Line 58: 2 forward slashes. function: assigns airport Code the value of the. Line 59: 2 forward slashes. new Airport Code parameter. Line 60: blank. Line 61: blank. Line 62: blank. Line 63 : 2 forward slashes. 7. 5 asterisks. Write this method 5 asterisks. Line 64: 2 forward slashes. Mutator method for the gates instance variable. Line 65: 2

forward slashes. method name: set Gates. Line 66: 2 forward slashes. return value: void. Line 67: 2 forward slashes. parameters: int new Gates. Line 68: 2 forward slashes. function: validates the new Gates parameter. Line 69: 2 forward slashes. if new Gates is greater than or equal to 0, Line 70: 2 forward slashes. sets gates to new Gates semicolon. Line 71: 2 forward slashes. otherwise, does not change the value of gates. Line 72: blank. Line 73: blank. Line 74: blank. Line 75: Right brace. 2 forward slashes. end of Airport class definition.

Back to Example

Program code. The code contains the following keywords: public, double, if, else, return, this, int, private, string, class. Line 1: Forward slash, asterisk. Auto class, Version 5. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Auto. Line 6: Left brace. Line 7: 2 forward slashes. instance variables. Line 8: private String model semicolon. 2 forward slashes. model of auto. Line 9: private int miles Driven semicolon. 2 forward slashes. number of miles driven. Line 10: private double gallons Of Gas semicolon. 2 forward slashes. number of gallons of gas. Line 11: blank. Line 12: 2 forward slashes. Default constructor. Line 13: 2 forward slashes. initializes model to double quotes unknown double quotes semicolon. Line 14: 2 forward slashes. miles Driven is auto initialized to 0. Line 15: 2 forward slashes. and gallons Of Gas to 0.0. Line 16: public Auto Left parentheses, right parentheses. Line 17:

Left brace. Line 18: model equals double quotes unknown double quotes semicolon. Line 19: Right brace. Line 20: blank. Line 21: 2 forward slashes. Overloaded constructor. Line 22: 2 forward slashes. allows client to set beginning values for. Line 23: 2 forward slashes. model, miles Driven, and gallons Of Gas. Line 24: public Auto left parentheses, String start Model. Line 25: int start Miles Driven. Line 26: double start Gallons Of Gas right parentheses. Line 27: Left brace. Line 28: model equals start Model semicolon. Line 29: set Miles Driven left parentheses, start Miles Driven right parentheses. semicolon. Line 30: set Gallons Of Gas left parentheses, start Gallons Of Gas right parentheses. semicolon. Line 31: Right brace. Line 32: blank. Line 33: 2 forward slashes. Accessor method. Line 34: 2 forward slashes. returns current value of model. Line 35: public String get Model Left parentheses, right parentheses. Line 36: Left brace. Line 37: return model semicolon. Line 38: Right brace. Line 39: blank. Line 40: 2 forward slashes. Accessor method. Line 41: 2 forward slashes. returns current value of miles Driven. Line 42: public int get Miles Driven Left parentheses, right parentheses. Line 43: Left brace. Line 44: return miles Driven semicolon. Line 45: Right brace. Line 46: blank. Line 47: 2 forward slashes. Accessor method. Line 48: 2 forward slashes. returns current value of gallons Of Gas. Line 49: public double get Gallons Of Gas Left parentheses, right parentheses. Line 50: Left brace. Line 51: return gallons Of Gas semicolon. Line 52: Right brace. Line 53: blank. Line 54: 2 forward slashes. Mutator method. Line 55: 2 forward slashes. allows client to set model. Line 56: public Auto set Model left parentheses, String model right parentheses. Line 57: Left brace. Line 58: this dot model equals model semicolon. Line 59: return this semicolon. Line 60: Right brace. Line 61: blank. Line 62: 2 forward slashes. Mutator method. Line 63: 2 forward slashes. allows client to set value of miles Driven semicolon. Line 64: 2 forward slashes. if new value is not less than 0. Line 65: public Auto set Miles Driven left parentheses, int miles Driven right parentheses. Line 66: Left brace. Line 67: if left parentheses, miles Driven right angled bracket, equals 0 right parentheses. Line 68: this dot miles Driven equals miles Driven semicolon. Line 69: return this semicolon. Line 70: Right brace. Line 71: blank. Line 72: 2 forward slashes. Mutator method. Line 73: 2 forward slashes. allows client to set value of gallons Of Gas semicolon. Line 74: 2 forward slashes. if new value is not less than 0.0. Line 75: public Auto set Gallons Of Gas left parentheses, double gallons Of Gas right parentheses. Line 76: Left brace. Line 77: System dot out dot print l n left parentheses, double quotes this is double quotes plus sign this right parentheses. semicolon. Line 78: if left parentheses, gallons Of Gas right angled bracket, equals 0.0 right parentheses. Line 79: this dot gallons Of Gas equals gallons Of Gas semicolon. Line 80: return this semicolon. Line 81: Right brace. Line 82: blank. Line 83: 2 forward slashes. Calculates miles per gallon. Line 84: 2 forward slashes. if no gallons of gas have been used, returns 0.0 semicolon. Line 85: 2 forward slashes. otherwise, returns miles per gallon. Line 86: 2 forward slashes. as miles Driven forward slash gallons Of Gas. Line 87: public double miles Per Gallon Left parentheses, right parentheses. Line 88: Left brace. Line 89: if left parentheses, gallons Of Gas right angled bracket, equals 0.0001 right parentheses. Line 90: return miles Driven forward slash gallons Of Gas semicolon. Line 91: else. Line 92: return 0.0 semicolon. Line 93: Right brace. Line 94: blank. Line 95: 2 forward slashes. Calculates money spent on gas. Line 96: 2 forward slashes. returns price per gallon times gallons of gas. Line 97: public double money Spent On Gas left parentheses, double price Per Gallon right parentheses. Line 98: Left brace. Line 99: return price Per Gallon asterisk gallons Of Gas semicolon. Line 100: Right brace. Line 101: Right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, new, int, double. Line 1: Forward slash, asterisk. Auto Client, Version 5. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Auto Client. Line 6: Left brace. Line 7: public static void main left parentheses, String left square bracket right square bracket a r g s right parentheses. Line 8: Left brace. Line 9: Auto car equals new Auto Left parentheses, right parentheses. semicolon. Line 10: System dot out dot print l n left parentheses, double quotes car is double quotes plus sign car right parentheses. semicolon. Line 11: car dot set Gallons Of Gas left parentheses, 3.4 right parentheses. set Miles Driven left parentheses, 67 right parentheses. semicolon. Line 12: blank. Line 13: int car Miles equals car dot get Miles Driven Left parentheses, right parentheses. semicolon. Line 14: double car Gallons equals car dot get Gallons Of Gas Left parentheses, right parentheses. semicolon. Line 15: blank. Line 16: System dot out dot print l n left parentheses, double quotes Miles driven is double quotes plus sign car Miles. Line 17: plus sign double quotes backslash n Gallons of gas is double quotes plus sign car Gallons right parentheses. semicolon. Line 18: Right brace. Line 19: Right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, String, return, new, Boolean, if, else, true, false. Line 1: Forward slash, asterisk. Auto class, Version 6. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: import java dot text dot Decimal Format semicolon. Line 6: blank. Line 7: public class Auto. Line 8: Left brace. Line 9: blank. Dot, dot, dot. Line 103: 2 forward slashes. to String: returns a String of instance variable values. Line 104: at symbol Override. Line 105: public String to String Left parentheses, right parentheses. Line 106: Left brace. Line 107: Decimal Format gallons Format equals new Decimal Format left parentheses, double Line quotes Hash 0.00 double quotes right parentheses. semicolon. Line 108:

return double quotes Model: double quotes plus sign model. Line 109: plus sign double quotes semicolon. miles driven: double quotes plus sign miles Driven. Line 110: plus sign double quotes semicolon. gallons of gas: double quotes. Line 111: plus sign gallons Format dot format left parentheses, gallons Of Gas right parentheses. Line semicolon. Line 112: Right brace. Line 113: blank. Line 114: 2 forward slashes. equals: returns true if fields of parameter object. Line 115: 2 forward slashes. are equal to fields in this object. Line 116: at symbol Override. Line 117: public boolean equals left parentheses, Object o right parentheses. Line 118: Left brace. Line 119: if left parentheses, o instance of Auto o b j Auto right parentheses. Line 120: Left brace. Line 121: if left parentheses, model dot equals left parentheses, o b j Auto dot model right parentheses. Line 122: 2 ampersands miles Driven equals equals o b j Auto dot miles Driven. Line 123: 2 ampersands Math dot a b s left parentheses, gallons Of Gas minus o b j Auto dot gallons Of Gas right parentheses. Line 124: left angled bracket, 0.0001 right parentheses. Line 125: return true semicolon. Line 126: else. Line 127: return false semicolon. Line 128: Right brace. Line 129: else. Line 130: return false semicolon. Line 131: Right brace. Line 132: Right brace.

Back to Example

The table has 3 columns labeled Operator, Syntax, and Operation. The row entries are as follows. Row 1. Operator: instance Of. Syntax: object Reference instance

Of Class Name. Operation: evaluates to true if object Reference is of Class Name type, semicolon, false otherwise.

Back to Table

Program code. The code contains the following keywords: public, class, static, void, String, new, if, else. Line 1: Forward slash, asterisk. Auto Client, version 6. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Auto Client. Line 6: Left brace. Line 7: public static void main left parentheses, String left square bracket right square bracket a r g s right parentheses. Line 8: Left brace. Line 9: Auto sporty equals new Auto left parentheses, double quotes Spyder double quotes , 0, 0.0 right parentheses. semicolon. Line 10: System dot out dot print l n, left parentheses, sporty dot to String Left parentheses, right parentheses. right parentheses. semicolon. Line 11: blank. Line 12: Auto compact equals new Auto left parentheses, double quotes Accent double quotes , 0, 0.0 right parentheses. semicolon. Line 13: System dot out dot print l n, Left parentheses, right parentheses. semicolon. Line 14: System dot out dot print l n left parentheses, compact right parentheses. semicolon. Line 15: blank. Line 16: if left parentheses, compact dot equals left parentheses, sporty right parentheses. right parentheses. Line 17: System dot out dot print l n, left parentheses, double quotes backslash n sporty and compact are equal double quotes right parentheses. semicolon. Line 18: else. Line

19: System dot out dot print l n, left parentheses, double quotes backslash n sporty and compact are not equal double quotes right parentheses. semicolon. Line 20: Right brace. Line 21: Right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, private, String, int, double, static. Line 1: Forward slash, asterisk. Auto class, Version 7. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: import java dot text dot Decimal Format, semicolon. Line 6: blank. Line 7: public class Auto. Line 8: Left brace. Line 9: 2 forward slashes. instance variables. Line 10: private String model semicolon. 2 forward slashes. model of auto. Line 11: private int miles Driven semicolon. 2 forward slashes. number of miles driven. Line 12: private double gallons Of Gas semicolon. 2 forward slashes. number of gallons of gas. Line 13: blank. Line 14: private static int count Autos equals 0 semicolon. 2 forward slashes. static class variable. Line 15: blank. Line 16: 2 forward slashes. Constructors. Line 17: 2 forward slashes. initializes model to double quotes unknown double quotes semicolon. Line 18: 2 forward slashes. miles Driven is auto initialized to 0. Line 19: 2 forward slashes. and gallons Of Gas to 0.0 semicolon. Line 20: 2 forward slashes. increments count Autos. Line 21: public Auto Left parentheses, right parentheses. Line 22: Left brace. Line 23: model equals double quotes unknown double quotes semicolon. Line

24: count Autos plus sign plus sign semicolon. 2 forward slashes. increment static count of Auto objects. Line 25: Right brace. Line 26: blank. Line 27: 2 forward slashes. allows client to set beginning values for. Line 28: 2 forward slashes. model, miles Driven, and gallons Of Gas semicolon. Line 29: 2 forward slashes. increments count Autos. Line 30: public Auto left parentheses, String start Model. Line 31: int start Miles Driven. Line 32: double start Gallons Of Gas right parentheses. Line 33: Left brace. Line 34: model equals start Model semicolon. Line 35: set Miles Driven left parentheses, start Miles Driven right parentheses. semicolon. Line 36: set Gallons Of Gas left parentheses, start Gallons Of Gas right parentheses. semicolon. Line 37: count Autos plus sign plus sign semicolon. 2 forward slashes. increment static count of Auto objects. Line 38: Right brace. Dot, dot, dot. Line 61: 2 forward slashes. Accessor method. Line 62: 2 forward slashes. returns count Autos. Line 63: public static int get Count Autos Left parentheses, right parentheses. Line 64: Left brace. Line 65: return count Autos semicolon. Line 66: Right brace. Dot, dot, dot. Line 145: Right brace.

Back to Example

The table has 2 columns labeled static Method and Non static Method. The row entries are as follows. Row 1. Access instance variables, question mark. static Method: no. Non static Method: yes. Row 2. Access static class variables, question mark. static Method: yes. Non static Method: yes. Row 3. Call static class methods, question mark. static Method: yes. Non static Method: yes. Row 4. Call non static instance methods, question mark. static Method: no. Non static Method: yes. Row 5. Use the reference this, question mark. static Method: no. Non static Method: yes.

Back to Table

Program code. The code contains the following keywords: public, class, static, void, String, new. Line 1: Forward slash, asterisk. Auto Client, Version 7. Line 2: Anderson, Franceschi. Line 3 Asterisk, Forward slash. Line 4: blank. Line 5: public class Auto Client. Line 6: Left brace. Line 7: public static void main left parentheses, String left square bracket right square bracket a r g s right parentheses. Line 8: Left brace. Line 9: System dot out dot print l n, left parentheses, double quotes Before instantiating an Auto object: double quotes. Line 10: plus sign double quotes backslash n the count of Auto objects is double quotes. Line 11: plus sign Auto dot get Count Autos, Left parentheses, right parentheses. right parentheses. semicolon. Line 12: blank. Line 13: Auto sporty equals new Auto left parentheses, double quotes Spyder double quotes , 0, 0.0 right parentheses. semicolon. Line 14: blank. Line 15: System dot out dot print l n left parentheses, double quotes backslash n After instantiating an Auto object: double quotes. Line 16: plus sign double quotes backslash n the count of Auto objects is double quotes. Line 17: plus sign Auto dot get Count Autos Left parentheses, right parentheses. right parentheses. semicolon. Line 18: Right brace. Line 19: Right brace.

Back to Example

Program code. The code contains the following keywords: public, class, private, int, return, this, if, new, void, Boolean, false. Line 1: Forward slash, asterisk. Rational class. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: public class Rational. Line 5: Left brace. Line 6: private int num semicolon. Line 7: private int d e n o m semicolon. Line 8: blank. Line 9: Forward slash, asterisk. asterisk. Line 10: asterisk Default constructor semicolon. num is set to 0, d e n o m is set to 1. Line 11: Asterisk, Forward slash. Line 12: public Rational Left parentheses, right parentheses. Line 13: Left brace. Line 14: d e n o m equals 1 semicolon. Line 15: Right brace. Line 16: blank. Line 17: Forward slash, asterisk. asterisk. Line 18: asterisk Overloaded constructor, d e n o m is set to 1. Line 19: asterisk at symbol param num an int, the initial value for num. Line 20: Asterisk, Forward slash. Line 21: public Rational left parentheses, int num right parentheses. Line 22 : Left brace. Line 23: d e n o m equals 1 semicolon. Line 24: set Num left parentheses, num right parentheses. semicolon. Line 25: Right brace. Line 26: blank. Line 27: Forward slash, asterisk. asterisk. Line 28: asterisk Overloaded constructor. Line 29: asterisk at symbol param num an int, the initial value for num. Line 30: asterisk at symbol param d e n o m an int, the initial value for d e n o m. Line

31: Asterisk, Forward slash. Line 32: public Rational left parentheses, int num, int d e n o m right parentheses. Line 33: Left brace. Line 34: this dot d e n o m equals 1 semicolon. Line 35: set Num left parentheses, num right parentheses. semicolon. Line 36: set D e n o m left parentheses, d e n o m right parentheses. semicolon. Line 37: Right brace. Line 38: blank. Line 39: Forward slash, asterisk. asterisk. Line 40: asterisk Accessor for num. Line 41: asterisk at symbol return an int, the value of num. Line 42: Asterisk, Forward slash. Line 43: public int get Num Left parentheses, right parentheses. Line 44: Left brace. Line 45: return num semicolon. Line 46: Right brace. Line 47: blank. Line 48: Forward slash, asterisk. asterisk. Line 49: asterisk Accessor for d e n o m. Line 50: asterisk at symbol return an int, the value of d e n o m. Line 51: Asterisk, Forward slash. Line 52: public int get D e n o m Left parentheses, right parentheses. Line 53: Left brace. Line 54: return d e n o m. semicolon. Line 55: Right brace. Line 56: blank. Line 57: Forward slash, asterisk. asterisk. Line 58: asterisk Mutator for num. Line 59: asterisk at symbol param num an int, the new value for num. Line 60: Asterisk, Forward slash. Line 61: public void set Num left parentheses, int num right parentheses. Line 62: Left brace. Line 63: this dot num equals num semicolon. Line 64: Right brace. Line 65: blank. Line 66: Forward slash, asterisk. asterisk. Line 67: asterisk Mutator for d e n o m. Line 68: asterisk at symbol param d e n o m an int, the new value for d e n o m. Line 69: Asterisk, Forward slash. Line 70: public void set D e n o m left parentheses, int d e n o m right parentheses. Line 71 Left brace. Line 72: if left parentheses, d e n o m exclamation mark equals 0 right parentheses. Line 73: Left brace. Line 74: this dot d e n o m equals d e n o m semicolon. Line 75: if left parentheses, this dot d e n o m left angled bracket, 0 right parentheses. Line 76: Left brace. Line 77: this dot d e n o m equals minus this dot d e n o m semicolon. Line 78: this dot n u m equals minus this dot num semicolon. Line 79: Right brace. Line 80: Right brace. Line 81: Right brace. Line 82: blank. Line 83: Forward slash, asterisk. asterisk. Line 84: asterisk add method. Line 85: asterisk at symbol param r a Rational. Line 86: asterisk at symbol return a Rational, the sum of this Rational and r. Line 87: Asterisk, Forward slash. Line 88: public Rational add left parentheses, Rational r right parentheses. Line 89: Left brace. Line 90: Rational result equals new Rational Left parentheses, right parentheses. semicolon. Line 91: result dot num equals num asterisk r dot d e n o m plus sign d e n o m asterisk r dot num semicolon. Line 92: result dot d e n o m equals d e n o m asterisk r dot d e n o m semicolon. Line 93: return result semicolon. Line 94: Right brace. Line 95: blank. Line 96: Forward slash, asterisk. asterisk. Line 97: asterisk multiply method. Line 98: asterisk at symbol param r a Rational. Line 99: asterisk at symbol return a Rational, the product of this Rational and r. Line 100: Asterisk, Forward slash. Line 101: public Rational multiply left parentheses, Rational r right parentheses. Line 102: Left brace. Line 103: Rational result equals new Rational Left parentheses, right parentheses. semicolon. Line 104: result dot num equals num asterisk r dot num semicolon. Line 105: result dot d e n o m equals d e n o m asterisk r dot d e n o m semicolon. Line 106 return result semicolon. Line 107 Right brace. Line 108: blank. Line 109: Forward slash, asterisk. asterisk. Line 110: asterisk simplify method semicolon. reduces this Rational to simplest terms. Line 111: Asterisk, Forward slash. Line 112: public void simplify Left parentheses, right parentheses. Line 113: Left brace. Line 114: int g equals Algebra dot g c d left parentheses, num, d e n o m right parentheses. semicolon. Line 115: num equals num forward slash g semicolon. Line 116: d e n o m equals d e n o m forward slash g semicolon. Line 117: Right brace. Line 118: blank. Line 119: Forward slash, asterisk. asterisk. Line 120: asterisk to String method. Line 121: asterisk at symbol return a String, the fraction for this Rational. Line 122: Asterisk, Forward slash. Line 123: public String to String Left parentheses, right parentheses. Line 124: Left brace. Line 125: String result equals double quotes double quotes plus sign num semicolon. Line 126: if left parentheses, num exclamation mark equals 0 2 ampersands d e n o m exclamation mark equals 1 right parentheses. Line 127: result plus sign equals double quotes forward slash double quotes plus sign d e n o m semicolon. Line 128: return result semicolon. Line 129: Right brace. Line 130: blank. Line 131: Forward slash, asterisk. asterisk. Line 132: asterisk equals method. Line 133: asterisk at symbol param o an Object. Line 134:

asterisk at symbol return a boolean, true if this Rational is equal to o. Line 135: asterisk false otherwise. Line 136: Asterisk, Forward slash. Line 137: public boolean equals left parentheses, Object o right parentheses. Line 138: Left brace. Line 139: if left parentheses, o instance of Rational o b j Rational right parentheses. Line 140: return num. asterisk o b j Rational dot d e n o m equals equals o b j Rational dot num asterisk d e n o m, semicolon. Line 141: else. Line 142: return false semicolon. Line 143: Right brace. Line 144: Right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, int, if, else, return, while, return. Line 1: Forward slash, asterisk. Algebra class. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: public class Algebra. Line 5: Left brace. Line 6: blank. Line 7: Forward slash, asterisk. asterisk. Line 8: asterisk g c d static method: computes and returns the g c d of a and b. Line 9: asterisk at symbol param a an int. Line 10: asterisk at symbol param b an int. Line 11: asterisk at symbol return an int, the gcd of a and b. Line 12 Asterisk, Forward slash. Line 13: public static int g c d left parentheses, int a, int b right parentheses. Line 14: Left brace. Line 15: 2 forward slashes. convert a and b to positive integers. Line 16: a equals Math dot max left parentheses, a, minus a right parentheses. semicolon. Line 17: b equals Math dot max left parentheses, b, minus b right parentheses. semicolon. Line 18: blank. Line 19: if left parentheses, b equals equals 0 right parentheses. Line 20: return a semicolon. Line 21: else. Line 22: Left brace. Line 23: int r equals a percentage sign b semicolon. Line 24: while left parentheses, r exclamation mark equals 0 right parentheses. Line 25: Left brace. Line 26: a equals b semicolon. Line 27: b equals r semicolon. Line 28: r equals a percentage sign b semicolon. Line 29: Right brace. Line 30: return b semicolon. Line 31: Right brace. Line 32: Right brace. Line 33: Right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, new, int. Line 1: Forward slash, asterisk. Rational Client class. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: public class Rational Client. Line 5: Left brace. Line 6: public static void main left parentheses, String left square bracket right square bracket a r g s right parentheses. Line 7: Left brace. Line 8: Rational r 1 equals new Rational left parentheses, 5, 3 right parentheses. semicolon. Line 9: System dot out dot print l n, left parentheses, double quotes r 1 equals double quotes plus sign r 1 right parentheses. semicolon. Line 10: Rational r 2 equals new Rational left parentheses, 7 right parentheses. semicolon. Line 11: System dot out dot print l n left parentheses, double quotes r 2 equals double quotes plus sign r 2 right parentheses. semicolon. Line 12: Rational r 3 equals new Rational Left parentheses, right parentheses. semicolon. Line 13: System dot out dot print l n left parentheses, double quotes r 3 equals double quotes plus sign r 3 right parentheses. semicolon. Line 14: Rational r 4 equals new Rational left parentheses, 5, minus 7 right parentheses. semicolon. Line 15: System dot out dot print l n left parentheses, double quotes r 4 equals double quotes plus sign r4 right parentheses. semicolon. Line 16: blank. Line 17: int g equals Algebra dot g c d left parentheses, 45, 75 right parentheses. semicolon. Line 18: System dot out dot print l n left parentheses, double quotes The g c d of 45 and 75 is double quotes plus sign g right parentheses. semicolon. Line 19: g equals Algebra dot g c d left parentheses, 3 asterisk 7 asterisk 11 asterisk 10, 7 asterisk 11 asterisk 23 right parentheses. semicolon. Line 20: System dot out dot print l n, left parentheses, double quotes g c d left parentheses, 7 asterisk 11 asterisk 13, 3 asterisk 7 asterisk 11 right parentheses. equals double quotes plus sign g right parentheses. semicolon. Line 21: blank. Line 22: Rational r 5 equals new Rational left parentheses, 25, 10 right parentheses. semicolon. Line 23: System dot out dot print l n, left parentheses, double quotes r 5 equals double quotes plus sign r 5 right parentheses. semicolon. Line 24: r 5 dot simplify Left parentheses, right parentheses. semicolon. Line 25: System dot out dot print l n left parentheses, double quotes In lowest terms, r 5 equals double quotes plus sign r 5 right parentheses. semicolon. Line 26: blank. Line 27: Rational r 6 equals new Rational left parentheses, 3 asterisk 7 asterisk 11 asterisk 10, minus 7 asterisk 11 asterisk 23 right parentheses. semicolon. Line 28: System dot out dot print I n left parentheses, double quotes r 6 equals double quotes plus sign r 6 right parentheses. semicolon. Line 29: r 6 dot simplify Left parentheses, right parentheses. semicolon. Line 30: System dot out dot print I n left parentheses, double quotes In lowest terms, r 6 equals double quotes plus sign r 6 right parentheses. semicolon. Line 31: blank. Line 32: Rational r 7 equals new Rational left parentheses, 5, 3 right parentheses. semicolon. Line 33: Rational r 8 equals new Rational left parentheses, 6, 4 right parentheses. semicolon. Line 34: Rational r 9 equals r 7 dot add left parentheses, r 8 right parentheses. semicolon. Line 35: r 9 dot simplify Left parentheses, right parentheses. semicolon. Line 36: System dot out dot print I n left parentheses, r 7 plus sign double quotes plus sign double quotes plus sign r 8 plus sign double quotes equals double quotes plus sign r 9 right parentheses. semicolon. Line 37: blank. Line 38: Rational r 10 equals r 7 dot multiply left parentheses, r 8 right parentheses. semicolon. Line 39: r 10 dot simplify Left parentheses, right parentheses. semicolon. Line 40: System dot out dot print I n left parentheses, r 7 plus sign double quotes asterisk double quotes plus sign r 8 plus sign double quotes equals double quotes plus sign r 10 right parentheses. semicolon. Line 41: Right brace. Line 42: Right brace.

Back to Example

Program code. The code contains the following keywords:

import, public, class, private, int, double, void, this, return. Line 1: Forward slash, asterisk. Sprite class. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: import java f x dot scene dot canvas. asterisk semicolon. Line 5: import java f x dot scene dot paint. asterisk semicolon. Line 6: blank. Line 7: public class Sprite. Line 8: Left brace. Line 9: private int s X semicolon. Line 10: private int s Y semicolon. Line 11: private double scale semicolon. Line 12: blank. Line 13: Forward slash, asterisk. asterisk default constructor. Line 14: asterisk s X equals s Y equals 0 semicolon. scale is set to 1. Line 15: Asterisk, Forward slash. Line 16: public Sprite Left parentheses, right parentheses. Line 17: Left brace. Line 18: scale equals 1 semicolon. Line 19: Right brace. Line 20: blank. Line 21: Forward slash, asterisk. overloaded constructor. Line 22: accepts values for starting x and y coordinates. Line 23: and scale. Line 24: Asterisk, Forward slash. Line 25: public Sprite left parentheses, int s X, comma, int s Y, double scale right parentheses. Line 26: Left brace. Line 27: set Coordinates left parentheses, s X, comma, s Y right parentheses. semicolon. Line 28: set Scale left parentheses, scale right parentheses. semicolon. Line 29: Right brace. Line 30: blank. Line 31: Forward slash, asterisk. set Coordinates. Line 32: asterisk accepts new values for starting x and y semicolon. Line 33: asterisk returns a reference to this object. Line 34: Asterisk, Forward slash. Line 35: public Sprite set Coordinates left parentheses, int s X, comma, int s Y, right parentheses. Line 36: Left brace. Line 37:

this dot s X equals s X semicolon. Line 38: this dot s Y equals s Y semicolon. Line 39: return this semicolon. Line 40: Right brace. Line 41: blank. Line 42: Forward slash, asterisk. mutator for scale. Line 43: asterisk returns a reference to this object. Line 44: Asterisk, Forward slash. Line 45: public Sprite set Scale left parentheses, double scale right parentheses. Line 46: Left brace. Line 47: this dot scale equals left parentheses, scale right angled bracket, 0 question mark scale colon this dot scale right parentheses. semicolon. Line 48: return this semicolon. Line 49: Right brace. Line 50: blank. Line 51: Forward slash, asterisk. draw method. Line 52: asterisk draws Sprite at current s X and s Y. Line 53: asterisk multiplying lengths by scale. Line 54: asterisk accepts Graphics Context for canvas. Line 55: Asterisk, Forward slash. Line 56: public void draw left parentheses, Graphics Context g c, right parentheses. Line 57: Left brace. Line 58: g c dot set Fill left parentheses, Color dot CORAL, right parentheses. semicolon. 2 forward slashes. body. Line 59: g c dot fill Oval left parentheses, s X, s Y plus sign 15 asterisk scale, 90 asterisk scale, 120 asterisk scale right parentheses. semicolon. Line 60: g c dot set Fill left parentheses, Color dot DARK GOLDEN ROD right parentheses. semicolon. 2 forward slashes. hat. Line 61: g c dot fill Rect left parentheses, s X plus sign 23 asterisk scale, s Y, 45 asterisk scale, 22 asterisk scale right parentheses. semicolon. Line 62: g c dot set Stroke left parentheses, Color dot DARK GOLDEN ROD right parentheses. semicolon. 2 forward slashes. hat brim. Line

63: g c dot set Line Width left parentheses, 3 right parentheses. semicolon. Line 64: g c dot stroke Line left parentheses, s X, s Y plus sign 23 asterisk scale. Line 65: s X plus sign 90 asterisk scale, s Y plus sign 23 asterisk scale right parentheses. semicolon. Line 66: g c dot set Fill left parentheses, Color dot CHOCOLATE right parentheses. semicolon. 2 forward slashes. eye. Line 67: g c dot fill Oval left parentheses, s X plus sign 60 asterisk scale, s Y plus sign 45 asterisk scale. Line 68: 18 asterisk scale, 12 asterisk scale right parentheses. semicolon. Line 69: g c dot set Fill left parentheses, Color dot DARK SALMON right parentheses. semicolon. 2 forward slashes. feet. Line 70: g c dot set Line Width left parentheses, 1 right parentheses. semicolon. Line 71: g c dot fill Oval left parentheses, s X plus sign 45 asterisk scale, s Y plus sign 125 asterisk scale. Line 72: 45 asterisk scale, 12 asterisk scale right parentheses. semicolon. Line 73: g c dot stroke Oval left parentheses, s X plus sign 45 asterisk scale, s Y plus sign 125 asterisk scale. Line 74: 45 asterisk scale, 12 asterisk scale right parentheses. semicolon. Line 75: g c dot fill Oval left parentheses, s X plus sign 27 asterisk scale, s Y plus sign 127 asterisk scale. Line 76: 45 asterisk scale, 12 asterisk scale right parentheses. semicolon. Line 77: g c dot stroke Oval left parentheses, s X plus sign 27 asterisk scale, s Y plus sign 127 asterisk scale. Line 78: 45 asterisk scale, 12 asterisk scale right parentheses. semicolon. Line 79 Right brace. Line 80 Right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, private, void, new, String. Line 1: Forward slash, asterisk. A client for the Sprite class. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: import java f x dot application dot Application semicolon. Line 6: import java f x dot scene dot canvas dot Graphics Context semicolon. Line 7: import java f x dot stage dot Stage semicolon. Line 8: blank. Line 9: public class Sprite Client extends Application. Line 10: Left brace. Line 11: private Sprite s 1, comma, s 2, comma, s 3 semicolon. Line 12: blank. Line 13: at symbol Override. Line 14: public void start left parentheses, Stage stage right parentheses. Line 15: Left brace. Line 16: Graphics Context g c equals J I Graphics Utility dot set Up Graphics left parentheses. Line 17: stage, double quotes Sprites double quotes, 700, 400 right parentheses. semicolon. Line 18: s 1 equals new Sprite left parentheses, 100, 50, 0.5 right parentheses. semicolon. Line 19: s 2 equals new Sprite left parentheses, 225, 100, 1 right parentheses. semicolon. Line 20: s 3 equals new Sprite Left parentheses, right parentheses. dot set Coordinates left parentheses, 400, 150 right parentheses, dot set Scale left parentheses, 1.5 right parentheses. semicolon. Line 21: blank. Line 22: s 1 dot draw left parentheses, g c right parentheses. semicolon. Line 23: s 2 dot draw left parentheses, g c right parentheses. semicolon. Line 24: s 3 dot draw left parentheses, g c right parentheses. semicolon. Line 25: Right brace. Line 26: blank. Line 27: public static void main left parentheses, String left square bracket, right square bracket a r g s right parentheses. Line 28: Left brace. Line 29: launch left parentheses, a r g s right parentheses. semicolon. Line 30: Right brace. Line 31: Right brace.

Back to Example

The table has 2 columns labeled Return type, and Method name and argument list. The row entries are as follows. Row 1. Return type: int. Method name and argument list: compare To, left parentheses, Enum e O b j, right parentheses, compares two enum objects and returns a negative number if this object is less than the argument, a positive number if this object is greater than the argument, and 0 if the two objects are the same. Row 2. Return type: boolean. Method name and argument list: equals, left parentheses, Object e O b j, right parentheses, returns true if this object is equal to the argument e O b j, semicolon, returns false otherwise. Row 3. Return type: int. Method name and argument list: ordinal, left parentheses, right parentheses, returns the numeric value of the enum object; by default; the value of the first object in the list is 0, the value of the second object is 1, and so on. Row 4. Return type: String. Method name and argument list: to String, left parentheses, right parentheses, returns the name of the enum constant. Row 5. Return type: enum. Method name and argument list: value Of, left parentheses, String enum Name, right parentheses, static method that returns the enum object whose name is the same as the String argument enum Name.

Back to Table

Program code. The code contains the following keywords: public, class, enum, static, void, String, if, else. Line 1: Forward slash, asterisk. Demonstration of enum. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Enum Demo. Line 6: Left brace. Line 7: public enum Days Left brace. SUN, comma, MON, comma, TUE, comma, WED, comma, THUR, comma, FRI, comma, SAT, Right brace. semicolon. Line 8: blank. Line 9: public static void main left parentheses, String left square bracket right square bracket a r g s right parentheses. Line 10: Left brace. Line 11: Days d 1, comma, d 2 semicolon. 2 forward slashes. declare two Days object references. Line 12: blank. Line 13: d 1 equals Days dot WED semicolon. Line 14: d 2 equals Days dot FRI semicolon. Line 15: blank. Line 16: System dot out dot print l n, left parentheses, double quotes Comparing objects using equals double quotes right parentheses. semicolon. Line 17: if left parentheses, d 1 dot equals left parentheses, d 2 right parentheses. right parentheses. Line 18: System dot out dot print l n left parentheses, d 1 plus sign double quotes equals double quotes plus sign d 2 right parentheses. semicolon. Line 19: else. Line 20: System dot out dot print l n left parentheses, d 1 plus sign double quotes does not equal double quotes plus sign d 2 right parentheses. semicolon. Line 21: blank. Line 22: System dot out dot print l n left parentheses, double quotes backslash n Comparing objects using compare To double quotes right parentheses. semicolon. Line 23: if, left parentheses, d 1 dot compare To, left parentheses, d 2 right parentheses. right angled bracket, 0 right parentheses. Line 24: System dot out dot print l n left parentheses, d 1 plus sign double quotes is greater than double quotes plus sign d 2 right parentheses. semicolon. Line 25: else if left parentheses, d 1 dot compare To left parentheses, d 2 right parentheses. left angled bracket, 0 right parentheses. Line 26: System dot out dot print l n left parentheses, d 1 plus sign double quotes is less than double quotes plus sign d 2 right parentheses. semicolon. Line 27: else. Line 28: System dot out dot print l n left parentheses, d 1 plus sign double quotes is equal to double quotes plus sign d 2 right parentheses. semicolon. Line 29: blank. Line 30: System dot out dot print l n left parentheses, double quotes backslash n Getting the ordinal value double quotes right parentheses. semicolon. Line 31: System dot out dot print l n left parentheses, double quotes The value of double quotes plus sign d 1 plus sign double quotes is double quotes. Line 32: plus sign d 1 dot ordinal Left parentheses, right parentheses. right parentheses. semicolon. Line 33: blank. Line 34: System dot out dot print l n left parentheses, double quotes backslash n Converting a String to an object double quotes right parentheses. semicolon. Line 35: Days day equals Days dot value Of left parentheses, double quotes MON double quotes right parentheses. semicolon. Line 36: System dot out dot print l n left parentheses, double quotes The value of day is double quotes plus sign day right parentheses. semicolon. Line 37: Right brace. Line 38: Right brace.

Back to Example

The code contains the following keywords: import, public, class, enum, static, void, String, new, switch, case. Line 1: Forward slash, asterisk. asterisk Specials of the Day. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner semicolon. Line 6: blank. Line 7: public class Daily Specials. Line 8: Left brace. Line 9: public enum Days Left brace. SUN, comma, MON, comma, TUE, comma, WED, comma, THUR, comma, FRI, comma, SAT, Right brace. semicolon. Line 10: blank. Line 11: public static void main left parentheses, String left square bracket right square bracket a r g s right parentheses. Line 12: Left brace. Line 13: Scanner scan equals new Scanner left parentheses, System.in right parentheses. semicolon. Line 14: blank. Line 15: System dot out dot print left parentheses, double quotes Enter a day backslash n double quotes. Line 16: plus sign double quotes left parentheses, Sun, comma, Mon, comma, Tue, comma, Wed, comma, Thur, comma, Fri, comma, Sat, right parentheses. right angled bracket, double quotes right parentheses. semicolon. Line 17: String input Day equals scan dot next Left parentheses, right parentheses. dot to Upper Case Left parentheses, right parentheses. semicolon. Line 18: Days day equals Days dot value Of left parentheses, input Day right parentheses. semicolon. Line 19: blank. Line 20: switch left parentheses, day right parentheses. Line 21: Left brace. Line 22: case MON, comma, WED minus right angled bracket, System dot out dot print l n left parentheses, double quotes The special for double quotes. Line 23: plus sign day plus sign double quotes is barbeque chicken double quotes right parentheses. semicolon. Line 24: case TUE,THUR minus right angled bracket, System dot out dot print l n left parentheses, double quotes The special for double quotes. Line 25: plus sign day plus sign double quotes is tacos double quotes right parentheses. semicolon. Line 26: case FRI minus right angled bracket, System dot out dot print l n left parentheses, double quotes The special for double quotes. Line 27: plus sign day plus sign double quotes is fish fillet double quotes right parentheses. semicolon. Line 28: case SUN, comma, SAT minus right angled bracket, System dot out dot print l n, left parentheses, double quotes Sorry, we're closed on double quotes. Line 29: plus sign day right parentheses. semicolon. Line 30: Right brace. Line 31: Right brace. Line 32: Right brace.

Back to Example

Program code. The code contains the following keywords: public, class, private, String, int, return, this. Line 1:

Forward slash, asterisk. Airport class. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Airport. Line 6: Left brace. Line 7: blank. Line 8: 2 forward slashes. instance variables. Line 9: private String airport Code semicolon. Line 10: private int gates semicolon. Line 11: blank. Line 12: 2 forward slashes. 1. 5 asterisks. Add a static class variable 5 asterisks. Line 13: 2 forward slashes. count Airports is an int. Line 14: 2 forward slashes. assign an initial value of 0. Line 15: blank. Line 16: blank. Line 17: 2 forward slashes. 2. 5 asterisks. Modify this method 5 asterisks. Line 18: 2 forward slashes. Default constructor, colon. Line 19: 2 forward slashes. method name: Airport. Line 20: 2 forward slashes. return value: none. Line 21: 2 forward slashes. parameters: none. Line 22: 2 forward slashes. function: sets the airport Code to an empty String. Line 23: 2 forward slashes. 5 asterisks. add 1 to count Airports class variable. Line 24: public Airport Left parentheses, right parentheses. Line 25: Left brace. Line 26: airport Code equals double quotes double quotes semicolon. Line 27: blank. Line 28: Right brace. Line 29: blank. Line 30: 2 forward slashes. 3. 5 asterisks. Modify this method 5 asterisks. Line 31: 2 forward slashes. Overloaded constructor, colon. Line 32: 2 forward slashes. method name: Airport. Line 33: 2 forward slashes. return value: none. Line 34: 2 forward slashes. parameters: a String airport code and an int start Gates. Line 35: 2 forward slashes. function: assigns airport Code the value of the. Line 36: 2 forward slashes. start Airport Code parameter semicolon. Line 37: 2 forward slashes. calls the set Gates method, comma. Line 38: 2 forward slashes. passing the start Gates parameter. Line 39: 2 forward slashes. 5 asterisks. add 1 to count Airports class variable. Line 40: public Airport left parentheses, String start Airport Code, int start Gates right parentheses. Line 41: Left brace. Line 42: airport Code equals start Airport Code semicolon. Line 43: set Gates left parentheses, start Gates right parentheses. semicolon. Line 44: blank. Line 45: Right brace. Line 46: blank. Line 47: 2 forward slashes. Accessor method for the airport Code instance variable. Line 48: 2 forward slashes. method name: get Airport Code. Line 49: 2 forward slashes. return value: String. Line 50: 2 forward slashes. parameters: none. Line 51: 2 forward slashes. function: returns airport Code. Line 52: public String get Airport Code Left parentheses, right parentheses. Line 53: Left brace. Line 54: return airport Code semicolon. Line 55: Right brace. Line 56: blank. Line 57: 2 forward slashes. Accessor method for the gates instance variable. Line 58: 2 forward slashes. method name: get Gates. Line 59: 2 forward slashes. return value: int. Line 60: 2 forward slashes. parameters: none. Line 61: 2 forward slashes. function: returns gates. Line 62: public int get Gates Left parentheses, right parentheses. Line 63: Left brace. Line 64: return gates semicolon. Line 65: Right brace. Line 66: blank. Line 67: 2 forward slashes. 4. 5 asterisks. Write this method 5 asterisks. Line 68: 2 forward slashes. Accessor method for the count Airports class variable. Line 69: 2 forward slashes. method name: get Count Airports. Line 70: 2 forward slashes. return value: int. Line 71: 2 forward slashes. parameters: none. Line 72: 2 forward slashes. function: returns count Airports. Line 73: blank. Line 74: blank. Line 75: blank. Line 76: blank. Line 77: 2 forward slashes. Mutator method for the airport Code instance variable. Line 78: 2 forward slashes. method name: set Airport Code. Line 79: 2 forward slashes. return value: Airport. Line 80: 2 forward slashes. parameters: String airport Code. Line 81: 2 forward slashes. function: assigns airport Code the value of the. Line 82: 2 forward slashes. airport Code parameter. Line 83: public Airport set Airport Code left parentheses, String airport Code right parentheses. Line 84: Left brace. Line 85: this dot airport Code equals airport Code semicolon. Line 86: return this semicolon. Line 87: Right brace. Line 88: blank. Line 89: 2 forward slashes. Mutator method for the gates instance variable. Line 90: 2 forward slashes. method name: set Gates. Line 91: 2 forward slashes. return value: Airport. Line 92: 2 forward slashes. parameters: int gates. Line 93: 2 forward slashes. function: validates the gates parameter. Line 94: 2 forward slashes. if gates is greater than 0, sets gates to gates semicolon. Line 95: 2 forward slashes. otherwise, does not change value of gates. Line 96: public Airport set Gates left parentheses, int gates right parentheses. Line 97: Left brace. Line 98: if left parentheses, gates right angled bracket, equals 0 right parentheses. Line 99: this dot gates equals gates semicolon. Line 100: return this semicolon. Line 101:

Right brace. Line 102: blank. Line 103: 2 forward slashes. 5. 5 asterisks. Write this method 5 asterisks. Line 104: 2 forward slashes. method name: to String. Line 105: 2 forward slashes. return value: String. Line 106: 2 forward slashes. parameters: none. Line 107: 2 forward slashes. function: returns a String that contains the airport Code. Line 108: 2 forward slashes. and gates. Line 109: blank. Line 110: blank. Line 111: blank. Line 112: blank. Line 113: blank. Line 114: blank. Line 115: 2 forward slashes. 6. 5 asterisks. Write this method 5 asterisks. Line 116: 2 forward slashes. method name: equals. Line 117: 2 forward slashes. return value: boolean. Line 118: 2 forward slashes. parameter: Airport object. Line 119: 2 forward slashes. function: returns true if airport Code. Line 120: 2 forward slashes. and gates in this object. Line 121: 2 forward slashes. are equal to those in the parameter object semicolon. Line 122: 2 forward slashes. returns false otherwise. Line 123: blank. Line 124: blank. Line 125: blank. Line 126: blank. Line 127: blank. Line 128: blank. Line 129: blank. Line 130: Right brace. 2 forward slashes. end of Airport class definition.

Back to Example

The table has 2 columns labeled File Name and Short Description. The row entries are as follows. Row 1. File Name: Auto dot h t m l. Short Description: Auto class documentation. Row 2. File Name: all classes hyphen index dot h t m l. Short Description: List of the classes with links. Row 3. File Name: index dot h t m l. Short

Description: Auto class documentation, left parentheses, with frames, right parentheses. Row 4. File Name: package hyphen summary dot h t m l. Short Description: Class hierarchy. Row 5. File Name: script. Short Description: Javascript file. Row 6. File Name: style sheet dot c s s. Short Description: Style sheet.

Back to Table

The table has 3 columns labeled Tag, Most Common Syntax, and Explanation. Row 1. Tag: at symbol, param. Most Common Syntax: at symbol, param variable Name description. Explanation: Adds a parameter to the parameter section. Row 2. Tag: at symbol. Most Common Syntax: at symbol return text. Explanation: Adds a description for the return type.

Back to Table

Program code. The code contains the following keywords: public, class, private, String, int, double, if, this, return. Line 1: Forward slash, asterisk. asterisk Simplified Auto Class with Javadoc comments. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Simplified Auto. Line 6: Left brace. Line 7: private String model semicolon. Line 8: private int miles Driven semicolon. Line 9: private double gallons Of Gas semicolon. Line 10: blank. Line 11: Forward slash, asterisk. asterisk. Line 12: asterisk Default constructor: left angled bracket, b r, right angled bracket. Line 13:

asterisk initializes model to double quotes unknown double quotes left angled bracket, b r, right angled bracket. Line 14: asterisk miles Driven are auto initialized to 0, and gallons Of Gas to 0.0. Line 15: Asterisk, Forward slash. Line 16: public Simplified Auto Left parentheses, right parentheses. Line 17: Left brace. Line 18: model equals double quotes unknown double quotes semicolon. Line 19: Right brace. Line 20: blank. Line 21: Forward slash, asterisk. asterisk. Line 22: asterisk Mutator method: left angled bracket, b r right angled bracket. Line 23: asterisk Allows client to set value of miles Driven left angled bracket, b r, right angled bracket. Line 24: asterisk, left angled bracket, strong right angled bracket, set Miles Driven left angled bracket, forward slash strong right angled bracket, does not change the value. Line 25: asterisk of left angled bracket, strong right angled bracket, miles Driven left angled bracket, forward slash strong right angled bracket, if new Miles Driven has negative value. Line 26: asterisk at symbol param miles Driven the new number of miles driven. Line 27: asterisk at symbol return a reference to this object. Line 28: Asterisk, Forward slash. Line 29: public Simplified Auto set Miles Driven left parentheses, int miles Driven right parentheses. Line 30: Left brace. Line 31: if left parentheses, miles Driven right angled bracket, 0 right parentheses. Line 32: this dot miles Driven equals miles Driven semicolon. Line 33: return this semicolon. Line 34: Right brace. Line 35: blank. Line 36: Forward slash, asterisk. asterisk. Line 37: asterisk Accessor method for miles Driven: left angled bracket, b r, right angled bracket. Line 38: asterisk at symbol return an int, the value of miles Driven. Line 39: Asterisk, Forward slash. Line 40: public int get Miles Driven Left parentheses, right parentheses. Line 41: Left brace. Line 42: return miles Driven semicolon. Line 43: Right brace. Line 44: Right brace.

Back to Example

Program code. The code contains the following keywords: double, for, int. Line 1: double daily Temp, semicolon. Line 2: double total dquals 0.0, semicolon. Line 3: for, left parentheses, int i equals 1, semicolon, i less than or equal to 365, semicolon, i plus, plus, right parentheses. Line 5: left brace. Line 6, indented once: System dot out dot print, left parentheses, double quotes, Enter a temperature, double quotes, right parentheses, semicolon. Line 7, indented once: daily Temp, equals, scan dot next Double, left parentheses, right parentheses, semicolon. Line 8, indented once: total, plus, equals, daily Temp, semicolon. Line 9: right brace. Line 10: double average, equals, total, forward slash, 365, semicolon.

Back to Code

The syntax is as follows. data type, left square bracket, right square bracket, array Name, semicolon.

Back to Code

The syntax is as follows. data type, array Name, left square bracket, right square bracket, semicolon.

Back to Code

String, left square bracket, right square bracket, c d Tracks, semicolon, 2 forward slashes, each element is a String object reference.

Back to Code

boolean, left square bracket, right square bracket, answers, semicolon, 2 forward slashes, each element is a boolean value.

Back to Code

Auto, left square bracket, right square bracket, cars, semicolon. 2 forward slashes, each element is an Auto object reference.

Back to Code

data type, left square bracket, right square bracket, array Name 1, comma, array Name 2, semicolon.

Back to Code

int, left square bracket, right square bracket, c s 101, comma, b i o 201, comma, h i s t 102, semicolon. 2 forward slashes, all elements are int values.

Back to Code

The syntax is as follows: array Name equals new data type, left square bracket, size, right square bracket, semicolon. The caption below the syntax is as follows: where size is an expression that evaluates to an integer and specifies the number of elements in the array.

Back to Code

Line 1: daily Temps equals new double, left square bracket, 365, right square bracket, semicolon, 2 forward slashes, daily Temps has 365 elements. Line 2: c d Tracks, equals, new String, left square bracket, 15, right square bracket, semicolon, 2 forward slashes, c d Tracks has 15 elements. Line 3: int number Of Questions equals 30, semicolon. Line 4: answers equals new boolean, left square bracket, number Of Questions, right square bracket, semicolon, 2 forward slashes, answers has 30 elements. Line 5: cars equals new Auto, left square bracket, 3, right square bracket, semicolon, 2 forward slashes, cars has 3 elements. Line 6: c s 101 equals new int, left square bracket, 5, right square bracket, semicolon. 2 forward slashes, c s 101 has 5 elements. Line 7: bio 201 equals new int, left square bracket, 4, right square bracket, semicolon. 2 forward slashes, bio 201 has 4 elements. Line 8: h i s t 102 equals new int, left square bracket, 6, right square bracket, semicolon, 2 forward slashes, h i s t 102 has 6 elements.

Back to Code

The table has 2 columns labeled Element Data Type and Initial Value. The row entries are as follows. Row 1. Element Data Type: double. Initial Value: 0.0. Row 2. Element Data Type: float. Initial Value: 0.0. Row 3. Element Data Type: int, long, short, byte. Initial Value: 0. Row 4. Element Data Type: char. Initial Value: null character, left square bracket, single quote, back slash, u, 0 0 0 0, single quote, right square bracket. Row 5. Element Data Type: boolean. Initial Value: false. Row 6. Element Data Type: object reference. Initial Value: null.

Back to Table

The syntax is as follows: data type, left square bracket, right square bracket, array Name, equals, new, data type, left square bracket, size, right square bracket, semicolon. The caption below the array is as follows: where size is an expression that evaluates to an integer and specifies the number of elements in the array.

Back to Code

double, left square bracket, right square bracket, daily Temps, equals, new double, left square bracket, 365, right square bracket, semicolon.

Back to Code

Line 1: double, left square bracket, right square bracket, daily Temps, semicolon. Line 2: daily Temps, equals, new double, left square bracket, 365, right square bracket,

semicolon.

Back to Code

String, left square bracket, right square bracket, c d Tracks, equals, new, String, left square bracket, 15, right square bracket, semicolon.

Back to Code

Line 1: String, left square bracket, right square bracket, c d Tracks, semicolon. Line 2: c d Tracks, equals, new, String, left square bracket, 15, right square bracket, semicolon.

Back to Code

The syntax is as follows: data type, left square bracket, right square bracket, array Name, equals, left brace, value 0, comma, value 1, comma, value 2, comma, dot, dot, dot, right brace, semicolon. The caption below the syntax is as follows: where value N is an expression that evaluates to the data type of the array and is the value to assign to the element at index N.

Back to Code

Line 1: int nine equals 9, semicolon. Line 2: int, left square bracket, right square bracket, odd Numbers equals, left brace, 1, comma, 3, comma, 5, comma, 7, comma, nine, comma, nine plus 2, comma, 13, comma, 15, comma, 17, comma, 19, right brace, semicolon.

Back to Code

Auto sports Car equals new Auto, left parentheses, double quotes, Ferrari, double quotes, comma, 0, comma, 0.0, right parentheses, semicolon. Line 2: Auto, left square bracket, right square bracket, cars equals, left brace, new Auto, left parentheses, double quotes, uppercase b, uppercase m, uppercase w, double quotes, comma, 100, comma, 15.0, right parentheses, comma, sports Car, comma, new Auto, left parentheses, right parentheses, right parentheses, right brace, semicolon.

Back to Code

The syntax is as follows: array Name, left square bracket, e x p, right square bracket. The caption below the syntax is as follows: where e x p is an expression that evaluates to an integer.

Back to Code

The syntax is as follows: array Name dot length.

Back to Code

The syntax is as follows: array Name, left square bracket, array Name dot length minus 1, right square bracket.

Back to Code

The table has 2 columns labeled Element and Syntax. The row entries are as follows. Row 1. Element: Element

0. Syntax: array Name, left square bracket, 0, right square bracket. Row 2. Element: Element i. Syntax: array Name, left square bracket, i, right square bracket. Row 3. Element: Last element. Syntax: array Name, left square bracket, array Name dot length minus 1, right square bracket.

Back to Table

Program code. The code contains the following keywords: public, class, static, void, String, double, new. Line 1: Forward slash. Asterisk. Array of Cell Phone Bills. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Cell Bills. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: 2 forward slashes. declare and instantiate the array. Line 10, indented twice: double, left square bracket, right square bracket, cell Bills, equals, new double, left square bracket, 6, right square bracket, semicolon. Line 11: blank. Line 12, indented twice: 2 forward slashes. assign values to array elements. Line 13, indented twice: cell Bills, left square bracket, 0, right square bracket, equals, 45.24, semicolon. Line 14, indented twice: cell Bills, left square bracket, 1, right square bracket, equals, 54.67, semicolon. Line 15, indented twice: cell Bills, left square bracket, 2, right square bracket, equals, 42.55. Line 16, indented twice: cell Bills, left square bracket, 3, right square bracket, equals, 44.61, semicolon. Line 17, indented twice: cell Bills, left square bracket, 4, right square bracket, equals, 65.29, semicolon. Line 18, indented twice: cell Bills, left square bracket, 5, right square bracket, equals, 49.75, semicolon. Line 19: blank. Line 20, indented twice: System dot out dot print l n, left parentheses, double quotes, The first monthly cell bill is, blank space, double quotes. Line 21, indented multiple times: plus, cell Bills, left square bracket, 0, right square bracket, right parentheses, semicolon. Line 22, indented multiple times: System dot out dot print l n, left parentheses, double quotes, The last monthly cell bill is, blank space, double quotes. Line 23, indented multiple times: plus, cell Bills, left square bracket, cell Bills dot length minus 1, right square bracket, right parentheses, semicolon. Line 24, indented once: right brace. Line 25: right brace.

Back to Example

Line 1: 2 forward slashes. Invalid indexes for the cell Bills array, 2 exclamation marks. Line 2: cell Bills, left square bracket, negative 1, right square bracket. 2 forward slashes, the lowest valid index is 0. Line 3: cell Bills, left square bracket, cell Bills dot length, right square bracket. 2 forward slashes, the highest valid index is. Line 4: 2 forward slashes, cell Bills dot length minus 1. Line 5: cell Bills, left square bracket, 150, right square bracket, 2 forward slashes, the highest valid index is 5.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, Auto, new. Line 1: Forward slash. Asterisk. Working with an Array of Objects. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Auto Array. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented once: 2 forward slashes. 1. instantiate cars array. Line 10, indented once: Auto, left square bracket, right square bracket, equals, new Auto, left square bracket, 3, right square bracket, semicolon. Line 11: blank. Line 12, indented once: 2 forward slashes. 2. instantiate Auto objects. Line 13, indented once: Auto sports Car, equals, new Auto, left parentheses, double quotes, Ferrari, double quotes, comma, 100, comma, 15.0, right parentheses, semicolon. Line 14, indented once: cars, left square bracket, 0, right square bracket, equals, sports Car, semicolon. 2 forward slashes, assign sports Car to element 0. Line 15, indented once: cars, left square bracket, 1, right square bracket, equals, new Auto, left parentheses, right parentheses, semicolon. 2 forward slashes, default Auto object. Line 16, indented once: 2 forward slashes, cars, left square bracket, 2, right square bracket, has not been instantiated and is null. Line 17: blank. Line 18, indented once: 2 forward slashes, call

Auto methods. Line 19, indented once: System dot out dot print l n, left parentheses, double quotes, cars, left square bracket, 0, right square bracket, is a, blank space, double quotes, plus, cars, left square bracket, 0, right square bracket, dot, get Model, left parentheses, right parentheses, right parentheses, semicolon. Line 20: blank. Line 21, indented once: Auto my Car, equals, cars, left square bracket, 1, right square bracket, semicolon. Line 22, indented once: System dot out dot print l n, left parentheses, double quotes, my Car has used, blank space, double quotes, plus, my Car dot get Gallons Of Gas, left parentheses, right parentheses. Line 23, indented multiple times: plus, double quotes, blank space, gallons of gas, double quotes, right parentheses, semicolon. Line 24: blank. Line 25, indented once: 2 forward slashes. attempt to call method when Auto object is not instantiated. Line 26, indented once: System dot out dot print l n, left parentheses, double quotes, cars, left square bracket, 2, right square bracket, is a, blank space, double quotes, plus, cars, left square bracket, 2, right square bracket, dot, get Model, left parentheses, right parentheses, right parentheses, semicolon. Line 27, indented once: right brace. Line 28: right brace.

Back to Example

The Auto class constructor summary is as follows. Auto, left parentheses, right parentheses, creates an Auto object with initial default values of double quotes, unknown, double quotes, 0, and 0.0. Auto, left parentheses, String model, comma, int miles Driven, comma, double gallons Of Gas, right parentheses, creates an Auto object with the initial values of model, miles Driven, and gallons Of Gas. The table summarizing the methods of the Auto Class has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: String. Method name and argument list: get Model, left parentheses, right parentheses, returns the value of model. Row 2. Return value: int. Method name and argument list: get Miles Driven, left parentheses, right parentheses, returns the value of miles Driven. Row 3. Return value: double. Method name and argument list: get Gallons Of Gas, left parentheses, right parentheses, returns the value of gallons Of Gas. Row 4. Return value: Auto. Method name and argument list: set Model, left parentheses, String model, right parentheses, mutator for model. Row 5. Return value: Auto. Method name and argument list: set Miles Driven, left parentheses, int miles Driven, right parentheses, mutator for miles Driven. Row 6. Return value: Auto. Method name and argument list: set Gallons Of Gas, left parentheses, double gallons Of Gas, right parentheses, mutator for gallons Of Gas. Row 7. Return value: double. Method name and argument list: miles Per Gallon, left parentheses, right parentheses, returns the gas mileage for this Auto object. Row 8. Return value: String. Method name and argument list: to String, left parentheses, right parentheses, returns a String representation of this object. Row 9. Return value:

boolean. Method name and argument list: equals, left parentheses, Object o b j, right parentheses, compares this Auto object to another object.

Back to Table

System dot out dot print l n, left parentheses, cell Bills, right parentheses, semicolon. 2 forward slashes, incorrect attempt to print array, exclamation mark.

Back to Code

for, left parentheses, int i equals 0, semicolon, i less than array Name dot length, semicolon, i plus, plus, right parentheses.

Back to Code

int i equals 0, semicolon.

Back to Code

i, plus, plus.

Back to Code

i less than array Name dot length.

Back to Code

array Name, left square bracket, i, right square bracket.

Back to Code

i less than or equal to array Name dot length.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, double, new, for, int. Line 1: Forward slash. Asterisk. Printing Array Elements. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Printing Array Elements. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: double, left square bracket, right square bracket, cell Bills, equals, new double, left square bracket, 6, right square bracket, semicolon. Line 10, indented twice: cell Bills, left square bracket, 0, right square bracket, equals, 45.24, semicolon. Line 11, indented twice: cell Bills, left square bracket, 1, right square bracket, equals, 54.67, semicolon. Line 12, indented twice: cell Bills, left square bracket, 2, right square bracket, equals, 42.55, semicolon. Line 13, indented twice: cell Bills, left square bracket, 3, right square bracket, equals, 44.61, semicolon. Line 14, indented twice: cells Bills, left square bracket, 4, right square bracket, equals, 65.29, semicolon. Line 15, indented twice: cell Bills, left square bracket, 5, right square bracket, equals, 49.75, semicolon. Line 16: blank. Line 17, indented twice: System dot out dot print l n, left parentheses, double quotes, Element, backslash, t, Value, double quotes, right parentheses, semicolon. Line 18, indented twice: for, left parentheses, int i equals 0, semicolon, i less than cell Bills dot length, semicolon, i plus, plus, right parentheses. Line 19, indented twice: left brace. Line 20, indented thrice: System dot out dot print l n, left parentheses, i plus double quotes, backslash t, double quotes, plus, cell Bills, left square bracket, i, right square bracket, right parentheses, semicolon. Line 21, indented twice: right brace. Line 22, indented once: right brace. Line 23: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, new, double, for, int. Line 1: Forward slash. Asterisk. Reading data into an array. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Reading Data Into An Array. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 12: blank. Line 13, indented twice: double, left square bracket, right square bracket, cell Bills, equals, new double, left square bracket, 6, right square bracket, semicolon. Line 14, indented twice: for, left parentheses, int i equals 0, semicolon, i less than cell Bills dot length, semicolon, i plus, plus, right parentheses. Line 15, indented twice: left brace. Line 16, indented thrice: System dot out dot print, left parentheses, double quotes, Enter bill amount for month, blank space, double quotes. Line 17, indented multiple times: plus, left parentheses, i, plus, 1, right parentheses, plus, double quotes, backslash, t, double quotes, right parentheses, semicolon. Line 18, indented thrice: cell Bills, left square bracket, i, right square bracket, equals, scan dot next Double, left parentheses, right parentheses, semicolon. 2 forward slashes, read current bill. Line 19, indented twice: right brace. Line 20, indented once: right brace. Line 21: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, double, new, for, int. Line 1: Forward slash. Asterisk. Summing Array Elements. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot text dot Number Format, semicolon. Line 6: blank. Line 7: public class Summing Array Elements. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: double, left square bracket, right square bracket, cell Bills, equals, new double, left square bracket, 6, right square bracket, semicolon. Line 12, indented twice: cell Bills, left square bracket, 0, right square bracket, equals, 45.24, semicolon. Line 13, indented twice: cell Bills, left square bracket, 1, right square bracket, equals, 54.67, semicolon. Line 14, indented twice: cell Bills, left square bracket, 2, right square bracket, equals, 42.55, semicolon. Line 15, indented twice: cell Bills, left square bracket, 3, right square bracket, equals, 44.61, semicolon. Line 16, indented twice: cell Bills, left square bracket, 4, right square bracket, equals, 65.29, semicolon. Line 17, indented twice: cell Bills, left square bracket, 5, right square bracket, equals, 49.75, semicolon. Line 18: blank. Line 19, indented twice: double total Bills, equals, 0.0, semicolon. 2 forward slashes. initialize total. Line 20, indented twice: for, left parentheses, int i equals 0, semicolon, i less than cell Bills dot length, semicolon, i plus, plus, right parentheses. Line 21, indented twice: left brace. Line 22, indented thrice: total Bills, plus, equals, cell Bills, left square bracket, i, right square bracket, semicolon. Line 23, indented twice: right brace. Line 24: blank. Line 25, indented twice: Number Format price Format, equals, Number Format dot get Currency Instance, left parentheses, right parentheses, semicolon. Line 26, indented twice: System dot out dot print l n, left parentheses, double quotes, Total for the bills, colon, blank space, double quotes, Total for the bills, colon, blank space, double quotes. Line 27, indented multiple times: plus, price Format dot format, left parentheses, total Bills, right parentheses, right parentheses, semicolon. Line 28, indented once: right brace. Line 29: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, double, new, int, for, if. Line 1: Forward slash. Asterisk. Finding the maximum array value. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot text dot Number Format, semicolon. Line 6: blank. Line 7: public class Max Array Value. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: double, left square bracket, right square bracket, cell Bills, equals, new double, left square bracket, 6, right square bracket, semicolon. Line 12, indented twice: cell Bills, left square bracket, 0, right square bracket, equals, 45.24, semicolon. Line 13, indented twice: cell Bills, left square bracket, 1, right square bracket, equals, 54.67, semicolon. Line 14, indented twice: cell Bills, left square bracket, 2, right square bracket, equals, 42.55, semicolon. Line 15, indented twice: cell Bills, left square bracket, 3, right square bracket, equals, 44.61, semicolon. Line 16, indented twice: cell Bills, left square bracket, 4, right square bracket, equals, 65.29, semicolon. Line 17, indented twice: cell Bills, left square bracket, 5, right square bracket, equals, 49.75, semicolon. Line 18: blank. Line 19, indented twice: int max Index, equals, 0, semicolon. 2 forward slashes. initialize to index of first element. Line 20, indented twice: for, left parentheses, int i equals 1, semicolon, i less than cell Bills dot length, semicolon, i plus, plus, right parentheses. Line 21, indented twice: left brace. Line 22, indented thrice: if, left parentheses, cell Bills, left square bracket, i, right square bracket, greater than, cell Bills, left square bracket, max Index, right square bracket, right parentheses. Line 23, indented 4 times: max Index, equals, i, semicolon. 2 forward slashes, save index of maximum value. Line 24, indented twice: right brace. Line 25: blank. Line 26, indented twice: Number Format price Format, equals, Number Format dot get Currency Instance, left parentheses, right parentheses, semicolon. Line 27, indented twice: System dot out dot print l n, left parentheses, double quotes, The highest bill, comma, blank space, double quotes. Line 28, indented multiple times: plus, price Format dot format, left parentheses, cell Bills, left square bracket, max Index, right square bracket, right parentheses. Line 29, indented multiple times: plus, double quotes, comma, was found at index, blank space, double quotes, plus, max Index, right parentheses, semicolon. Line 30, indented once: right brace. Line 31: right brace.

Back to Example

double, left square bracket, right square bracket, bills

Backup equals new double, left square bracket, 6, right square bracket, semicolon.

Back to Code

bills Backup equals cell Bills, semicolon, 2 forward slashes, incorrect attempt to copy array elements, exclamation mark.

Back to Code

bills Backup, left square bracket, 4, right square bracket, equals, 38.00, semicolon.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, double, new, for, int. Line 1: Forward slash. Asterisk. Copying Array Elements to Another Array. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Copying Array Elements. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: double, left square bracket, right square bracket, cell Bills, equals, left brace, 45.24, comma, 54.67, comma, 42.55, comma, 44.61, comma, 65.29, comma, 49.75, right brace, semicolon. Line 10: blank. Line 11, indented twice: double bills Backup, left square bracket, right square bracket, equals, new, double, left square bracket, cell Bills dot length, right square bracket, semicolon. Line 12, indented twice: for, left parentheses, int i equals 0, semicolon, i less than cell Bills dot length, semicolon, i, plus, plus, right parentheses. Line 13, indented twice: left brace. Line 14, indented thrice: bills Backup, left square bracket, i, right square bracket, equals, cell Bills, left square bracket, i, right square bracket, semicolon. 2 forward slashes, copy each element. Line 15, indented twice: right brace. Line 16: blank. Line 17, indented twice: bills Backup, left square bracket, 4, right square bracket, equals, 38.00, semicolon. 2 forward slashes, change value in bills Backup. Line 18: blank. Line 19, indented twice: System dot out dot print l n, left parentheses, double quotes, cell Bills, backslash, n, Element, backslash, t, Value, blank space, double quotes, right parentheses, semicolon. Line 20, indented twice: for, left parentheses, int i equals 0, semicolon, i less than cell Bills dot length, semicolon, i plus, plus, right parentheses. Line 21, indented twice: left brace. Line 22, indented thrice: System dot out dot print l n, left parentheses, i plus, double quotes, backslash, t, double quotes, plus, cell Bills, left square bracket, i, right square bracket, right parentheses, semicolon. Line 23, indented twice: right brace. Line 24: blank. Line 25, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, bills Backup, backslash, n, Element, backslash, t, Value, blank space, double quotes, right parentheses, semicolon. Line 26, indented twice: for, left parentheses, int i equals 0, semicolon, i less than bills

Backup dot length, semicolon, i plus, plus, right parentheses. Line 27, indented twice: left brace. Line 28, indented twice: System dot out dot print l n, left parentheses, i plus, double quotes, backslash t, double quotes, plus, bills Backup, left square bracket, i, right square bracket, right parentheses, semicolon. Line 29, indented twice: right brace. Line 30, indented once: right brace. Line 31: right brace.

Back to Example

cell Bills, equals, new double, left square bracket, 12, right square bracket, semicolon.

Back to Code

Program code. The code contains the following keywords: double, new, for, int, null. Line 1: double, left square bracket, right square bracket, temp, equals, new double, left square bracket, 12, right square bracket, semicolon, 2 forward slashes, instantiate new array. Line 2: blank. Line 3: 2 forward slashes, copy all elements from cell Bills to temp. Line 4: for, left parentheses, int i equals 0, semicolon, i less than cell Bills dot length, semicolon, i, plus, plus, right parentheses. Line 5: left brace. Line 6, indented once: temp, left square bracket, i, right square bracket, equals, cell Bills, left square bracket, i, right square bracket, semicolon, 2 forward slashes, copy each element. Line 7: right brace. Line 8: cell Bills equals temp, semicolon, 2 forward slashes, assign temp to cell Bills.

Line 9: temp equals null, semicolon, 2 forward slashes, temp no longer points to cell Bills.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, double, boolean, true, if, false, else, for, int, else. Line 1: Forward slash. Asterisk. Comparing Arrays of primitive data types. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Comparing Arrays. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: double, left square bracket, right square bracket, cell Bills 1, equals, left brace, 45.24, comma, 54.67, comma, 42.55, comma, 44.61, comma, 65.29, comma, 49.75, right brace, semicolon. Line 10, indented twice: double, left square bracket, right square bracket, cell Bills 2, equals, left brace, 45.24, comma, 54.67, comma, 41.99, comma, 44.61, comma, 65.29, comma, 49.75, right brace, semicolon. Line 11: blank. Line 12, indented twice: boolean is Equal, equal, true, semicolon. Line 13, indented twice: if, left parentheses, cell Bills 1 dot length, exclamation mark, equals, cell Bills 2 dot length, right parentheses. Line 14, indented twice: left brace. Line 15, indented thrice: is Equal, equals, false, semicolon. 2 forward slashes, arrays are not the same size. Line 16, indented once: right brace. Line 17, indented twice: else.

Line 18, indented twice: left brace. Line 19, indented thrice: for, left parentheses, int i equals 0, semicolon, i less than cell Bills 1 dot length, 2 ampersands, is Equal, semicolon, i plus, plus, right parentheses. Line 20, indented thrice: left brace. Line 21, indented 4 times: if, left parentheses, Math dot a b s, left parentheses, cell Bills, left square bracket, i, right square bracket, minus, cell Bills 2, left square bracket, i, right square bracket, right parentheses, greater than, 0.001, right parentheses. Line 22, indented 4 times: left brace. Line 23, indented 5 times: is Equal, equals, false, semicolon. 2 forward slashes, elements are not equal. Line 24, indented 4 times: right brace. Line 25, indented thrice: right brace. Line 26, indented twice: right brace. Line 27: blank. Line 28, indented twice: if, left parentheses, is Equal, right parentheses. Line 29, indented thrice: System dot out dot print l n, left parentheses, double quotes, cell Bills 1 and cell Bills 2 are equal, double quotes, right parentheses, semicolon. Line 30, indented twice: else. Line 31, indented thrice: System dot out dot print l n, left parentheses, double quotes, cell Bills 1 and cell Bills 2 are not equal, double quotes, right parentheses, semicolon. Line 32, indented once: right brace. Line 33: right brace.

Back to Example

if, left parentheses, int Array 1, left square bracket, i, right square bracket, exclamation mark, equals sign, int Array 2, left square bracket, i, right square bracket, right parentheses.

Back to Code

if, left parentheses, exclamation mark, cars 1, left square bracket, i, right square bracket, dot, equals, left parentheses, cars 2, left square bracket, i, right square bracket, right parentheses, right parentheses.

Back to Code

if, left parentheses, cell Bills, 2 equals sign, bills Backup, right parentheses.

Back to Code

if, left parentheses, cell Bills dot equals, left parentheses, bills Backup, right parentheses, right parentheses.

Back to Code

Program code. The code contains the following keywords: import, public, class, extends, final, int, private, double, String, void, for, static. Line 1: Forward slash. Asterisk. A bar chart application. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java f x dot application dot Application, semicolon. Line 6: import java f x dot stage dot Stage, semicolon. Line 7: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 8: import java f x dot scene dot paint dot Color, semicolon. Line 9: blank. Line 10: public class Bar Chart Application extends Application. Line 11: left brace. Line 12, indented once:

final int LEFT underscore MARGIN, equals, 35, semicolon. 2 forward slashes, starting x coordinate. Line 13, indented once: final int BASE underscore Y underscore BAR equals 150, semicolon. 2 forward slashes, bottom of the bars. Line 14, indented once: final int BASE underscore Y underscore VALUE, equals, 175, semicolon, 2 forward slashes, bottom of the values. Line 15, indented once: final int BAR underscore WIDTH, equals, 30, semicolon. 2 forward slashes, width of each bar. Line 16, indented once: final int SPACE underscore BETWEEN underscore BARS, equals, 10, semicolon. 2 forward slashes, pixels between bars. Line 17: blank. Line 18, indented once: private double, left square bracket, right square bracket, cell Bills. Line 19, indented twice: equals, left brace, 45.24, comma, 54.67, comma, 42.55, comma, 44.61, comma, 65.29, comma, 49.75, right brace, semicolon. Line 20, indented once: private String, left square bracket, right square bracket, months. Line 21, indented twice: equals, left brace, double quotes, March, double quotes, comma, double quotes, April, double quotes, comma, double quotes, May, double quotes, comma, double quotes, June, double quotes, comma, double quotes, July, double quotes, comma, double quotes, August, double quotes, right parentheses, semicolon. Line 22: blank. Line 23, indented once: At symbol, Override. Lien 24, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 25, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 25, indented once:

left brace. Line 26, indented twice: 2 forward slashes, set up window title and size. Line 27, indented twice: Graphics Context g c, equals, J I Graphics Utility dot set Up Graphics, left parentheses. Line 28, indented multiple times: stage, comma, double quotes, Bar char, double quotes, comma, 350, comma, 200, right parentheses, semicolon. Line 29: blank. Line 30, indented twice: g c dot set Fill, left parentheses, Color dot BLUE, right parentheses, semicolon. 2 forward slashes, bars will be blue. Line 31, indented twice: int x Start, equals, LEFT underscore MARGIN, semicolon. 2 forward slashes, x value for first bar. Line 32: blank. Line 33, indented twice: for, left parentheses, int i equals 0, semicolon, i less than cell Bills dot length, semicolon, i plus, plus, right parentheses. Line 34, indented twice: left brace. Line 35, indented thrice: g c dot fill Rect, left parentheses, x Start, comma, BASE underscore Y underscore BAR minus cell Bills, left square bracket, i, right square bracket, comma. Line 36, indented multiple times: BAR underscore WIDTH, coma, cell Bills, left square bracket, i, right square bracket, right parentheses, semicolon. Line 37: blank. Line 38, indented thrice: g c dot fill Text, left parentheses, Double dot to String, left parentheses, cell Bills, left square bracket, i, right square bracket, right parentheses, comma. Line 39, indented multiple times: x Start, comma, BASE underscore Y underscore VALUE, right parentheses, semicolon. Line 40: blank. Line 41, indented thrice: 2 forward slashes. move to starting x value for next bar. Line 42, indented thrice: x Start, plus, equals, BAR underscore WIDTH, plus, SPACE underscore BETWEEN underscore BARS, semicolon. Line 43, indented twice: right brace. Line 44, indented once: right brace. Line 45: blank. Line 46, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 47, indented once: left brace. Line 48, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 49, indented once: right brace. Line 50: right brace.

Back to Example

Line 1: 2 forward slashes 5 asterisks 1. The first method has been coded as an example. Line 2: Forward slash, asterisk. Asterisk. Fills the array with random numbers between 50 and 80. Line 3: Asterisk. The instance variable a r r is the integer array. Line 4: Asterisk. to be filled with values. Line 5: Asterisk, Forward slash. Line 6: public void fill Values left parentheses, right parentheses. Line 7: left brace. Line 8: Random rand equals new Random left parentheses, right parentheses. semicolon. Line 9: for left parentheses int i equals 0 semicolon. i left angled bracket a r r dot length semicolon. i plus, plus. Right parentheses. Line 10: left brace. Line 11: a r r left square bracket i right square bracket equals rand dot next Int, left parentheses 31 Right parentheses. plus sign, 50 semicolon. Line 12: animate left parentheses i, a r r left square bracket i right square bracket Right parentheses. semicolon. 2 forward slashes needed to create visual

feedback. Line 13: right brace. Line 14: right brace. Line 15: 2 forward slashes end of fill Values method. Line 16: 2 forward slashes 5 asterisks 2. student writes this method. Line 17: Forward slash, asterisk. Asterisk. Prints the array to the console with elements separated. Line 18: Asterisk. by a space. Line 19: Asterisk. The instance variable a r r is the integer array to be printed. Line 20: Asterisk, Forward slash. Line 21: public void print Array left parentheses, right parentheses. Line 22: left brace. Line 23: 2 forward slashes Note: to animate the algorithm, put this method call as the Line 24: 2 forward slashes last statement in your for loop: Line 25: 2 forward slashes animate left parentheses i, a r r left square bracket i right square bracket Right parentheses. semicolon. Line 26: 2 forward slashes where i is the index of the current array element. Line 27: 2 forward slashes Write your code here: Line 28: right brace. 2 forward slashes end of print Array method. Line 29: 2 forward slashes 5 asterisks 3. student writes this method. Line 30: Forward slash, asterisk. Asterisk. Sets all the elements in the array to parameter value. Line 31: Asterisk. The instance variable a r r is the integer array to be processed. Line 32: Asterisk. at symbol param value the value to which to set the array elements. Line 33: Asterisk, Forward slash. Line 34: public void set Values left parentheses int value Right parentheses. Line 35: left brace. Line 36: 2 forward slashes Note: to animate the algorithm, put this method call as the. Line 37: 2 forward slashes last statement in your for loop. Line 38: 2 forward slashes animate, left parentheses I, a r r left square bracket i right square bracket Right parentheses. semicolon. Line 39: 2 forward slashes where i is the index of the current array element. Line 40: 2 forward slashes Write your code here: Line 41: right brace. 2 forward slashes end of set Values method. Line 42: 2 forward slashes 5 asterisks 4. student writes this method. Line 43: Forward slash, asterisk. Asterisk. Counts number of elements equal to parameter value. Line 44: Asterisk. The instance variable arr is the integer array to be processed. Line 45: Asterisk. at symbol param value the value to count. Line 46: Asterisk. at symbol return the number of elements equal to value. Line 47: Asterisk, Forward slash. Line 48: public int count Frequency left parentheses int value Right parentheses. Line 49: left brace. Line 50: 2 forward slashes Note: to animate the algorithm, put this method call as the. Line 51: 2 forward slashes last statement in your for loop. Line 52: 2 forward slashes animate, left parentheses i, count Right parentheses. semicolon. Line 53: 2 forward slashes where i is the index of the current array element. Line 54: 2 forward slashes count is the variable holding the frequency. Line 55: 2 forward slashes Write your code here: Line 56: return 0 semicolon. 2 forward slashes replace this line with your return statement. Line 57: right brace. 2 forward slashes end of count Frequency method. Line 58: 2 forward slashes 5 asterisks 5. student writes this method. Line 59: Forward slash, asterisk. Asterisk. Finds and returns the minimum value in a r r. Line 60: Asterisk. The instance variable a r r is the integer array to be processed. Line 61: Asterisk. at symbol return the minimum value found in a r r. Line 62: Asterisk, Forward slash. Line 63: public int find Minimum left parentheses, right parentheses. Line 64: left brace. Line 65: 2 forward slashes Note: to animate the algorithm, put this method call as the. Line 66: 2 forward slashes last statement in your for loop. Line 67: 2 forward slashes animate, left parentheses i, minimum Right parentheses. semicolon. Line 68: 2 forward slashes where i is the index of the current array element. Line 69: 2 forward slashes minimum is the variable holding the minimum. Line 70: 2 forward slashes Write your code here: Line 71: return 0 semicolon. 2 forward slashes replace this line with your return statement. Line 72: right brace. 2 forward slashes end of find Minimum method. Line 73: 2 forward slashes End of student code.

Back to Example

Program code. The code uses the following keywords: for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i less than a r r dot length, semicolon, i plus, plus, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, a r r, left square bracket, i, right square bracket, right parentheses, semicolon. Line 3, indented once: animate, left parentheses, i, right parentheses, semicolon. 2 forward slashes, this statement is outside the for loop.

Back to Code

Program code. The code uses the following keywords: for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i less than a r r dot length, semicolon, i plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: System dot out dot print l n, left parentheses, a r r, left square bracket, i, right square bracket, right parentheses, semicolon. Line 4, indented once: animate, left parentheses, i, right parentheses, semicolon. Line 5: right brace.

Back to Code

for, left parentheses, int i equals 0, semicolon, i less than a r r dot length, semicolon, i plus, plus, right parentheses.

Back to Code

The syntax is as follows: access Modifier return Type method Name, left parentheses, data Type, left square bracket, right square bracket, array Name, right parentheses.

Back to Code

The syntax is as follows: access Modifier data Type, left square bracket, right square bracket, method Name, left parentheses, parameter List, right parentheses.

Back to Code

The syntax is as follows: method Name, left parentheses, array Name, right parentheses.

Back to Code

Program code. The code contains the following keywords: import, public, class, final, int, private, String, double, new, this, for, return, Boolean, if, else, true, false. Line 1: Forward slash, asterisk. Asterisk. Cell Phone class. Line 2: Asterisk. Anderson, comma, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: import java dot text dot Decimal Format semicolon. Line 6: blank. Line 7: public class Cell Phone. Line 8: left brace. Line 9: public final int MONTHS equals 6 semicolon. 2 forward slashes default number of months. Line 10: private String phone Number semicolon. Line 11: private double left square bracket, right square bracket cell Bills semicolon. Line 12: blank. Line 13: Forward slash, asterisk. Asterisk. Default constructor. Line 14: Asterisk. creates cell Bills with MONTHS elements. Line 15: Asterisk, Forward slash. Line 16: public Cell Phone left parentheses, right parentheses. Line 17: left brace. Line 18: phone Number equals double quotes double quotes semicolon. Line 19: cell Bills equals new double left square bracket MONTHS right square bracket semicolon. Line 20 right brace. Line 21: blank. Line 22: Forward slash, asterisk. Asterisk. Constructor. Line 23: Asterisk. at symbol param phone Number cell phone number. Line 24: Asterisk. at symbol param cell Bills array of monthly bills. Line 25: Asterisk, Forward slash. Line 26: public Cell Phone, left parentheses, String phone Number, double left square bracket right square bracket cell Bills Right parentheses.

Line 27: left brace. Line 28: this dot phone Number equals phone Number semicolon. Line 29: blank. Line 30: 2 forward slashes instantiate array with same length as parameter. Line 31: this dot cell Bills equals new double left square bracket cell Bills dot length right square bracket semicolon. Line 32: blank. Line 33: 2 forward slashes copy parameter array to cell Bills array. Line 34: for left parentheses int i equals 0 semicolon. i left angled bracket cell Bills dot length semicolon. i plus, plus. Right parentheses. Line 35: left brace. Line 36: this dot cell Bills left square bracket i right square bracket equals cell Bills left square bracket i right square bracket semicolon. Line 37: right brace. Line 38: right brace. Line 39: blank. Line 40: Forward slash, asterisk. Asterisk. Accessor for the phone number. Line 41: Asterisk. at symbol return the phone number. Line 42: Asterisk, Forward slash. Line 43: public String get Phone Number left parentheses, right parentheses. Line 44: left brace. Line 45: return phone Number semicolon. Line 46: right brace. Line 47: blank. Line 48: Forward slash, asterisk. Asterisk. Accessor for the cell phone bills. Line 49: Asterisk. at symbol return copy of cell Bills array. Line 50: Asterisk, Forward slash. Line 51: public double left square bracket right square bracket get Cell Bills left parentheses, right parentheses. Line 52: left brace. Line 53: double left square bracket right square bracket temp equals new double left square bracket cell Bills dot length right square bracket semicolon. Line 54: for left parentheses int i equals 0 semicolon. i left angled bracket cell Bills dot length semicolon. i plus, plus. Right parentheses. Line 55: left brace. Line 56: temp left square bracket i right square bracket equals cell Bills left square bracket i right square bracket semicolon. Line 57: right brace. Line 58: return temp semicolon. Line 59: right brace. Line 60: blank. Line 61: Forward slash, asterisk. Asterisk. Calculates total of all cell phone bills. Line 62: Asterisk. at symbol return total of all elements in cell Bills array. Line 63: Asterisk, Forward slash. Line 64: public double c a l c Total Bills left parentheses, right parentheses. Line 65: left brace. Line 66: double total equals 0.0 semicolon. 2 forward slashes initialize total to 0.0. Line 67: blank. Line 68: for left parentheses int i equals 0 semicolon. i left angled bracket cell Bills dot length semicolon. i plus, plus. Right parentheses. Line 69: left brace. Line 70: total plus sign equals cell Bills left square bracket i right square bracket semicolon. 2 forward slashes add current element to total. Line 71: right brace. Line 72: return total semicolon. Line 73: right brace. Line 74: blank. Line 75: Forward slash, asterisk. Asterisk. Finds a maximum bill. Line 76: Asterisk. at symbol return largest value in cell Bills array. Line 77: Asterisk, Forward slash. Line 78: public double find Maximum Bill left parentheses, right parentheses. Line 79: left brace. Line 80: double max equals cell Bills left square bracket 0 right square bracket semicolon. 2 forward slashes assume first element is max. Line 81: blank. Line 82: for left parentheses int i equals 1 semicolon. i left angled bracket cell Bills dot length semicolon. i plus, plus. Right parentheses. Line 83: left brace. Line 84: if left parentheses cell Bills left square bracket i right square bracket greater than max Right parentheses. Line 85: max equals cell Bills left square bracket i right square bracket semicolon. 2 forward slashes save new maximum. Line 86: right brace. Line 87 return max semicolon. Line 88 right brace. Line 89: blank. Line 90: Forward slash, asterisk. Asterisk. Returns printable version of CellPhone object. Line 91: Asterisk. at symbol return phone number plus each month's bill. Line 92: Asterisk, Forward slash. Line 93: at symbol Override. Line 94: public String to String left parentheses, right parentheses. Line 95: left brace. Line 96: String return Value equals phone Number plus sign double quotes backslash n double quotes semicolon. Line 97: Decimal Format money equals new Decimal Format, left parentheses double quotes dollar sign, 2 hash signs, 0.00 double quotes Right parentheses. semicolon. Line 98: for left parentheses int i equals 0 semicolon. i left angled bracket cell Bills dot length semicolon. i plus, plus. Right parentheses. Line 99: left brace. Line 100: return Value plus sign equals money dot format left parentheses cell Bills left square bracket i right square bracket Right parentheses. plus sign double quotes backslash t double quotes semicolon. Line 101: right brace. Line 102: return Value plus sign equals double quotes backslash n double quotes semicolon. Line 103: blank. Line 104: return return Value semicolon. Line 105: right brace. Line 106: blank. Line 107: Forward slash, asterisk. Asterisk. Compares two Cell Phone objects for equality. Line 108: Asterisk. at symbol param c Cell Phone object. Line 109: Asterisk. at symbol return true if objects are equal semicolon. false, otherwise. Line 110: Asterisk, Forward slash. Line 111: at symbol Override. Line 112: public boolean equals left parentheses Object c Right parentheses. Line 113: left brace. Line 114: if left parentheses, exclamation mark, left parentheses c instance of Cell Phone Right parentheses. Right parentheses. Line 115: return false semicolon. Line 116: else. Line 117: left brace. Line 118: Cell Phone o b j C P equals left parentheses Cell Phone Right parentheses. c semicolon. Line 119: if left parentheses ! left parentheses phone Number dot equals left parentheses o b j C P dot phone Number Right parentheses. Right parentheses. Right parentheses. Line 120: return false semicolon. Line 121: blank. Line 122: if left parentheses cell Bills dot length, exclamation mark, equals o b j C P dot cell Bills dot length Right parentheses. Line 123: return false semicolon. 2 forward slashes arrays are not the same length. Line 124: blank. Line 125: for left parentheses int i equals 0 semicolon. i left angled bracket cell Bills dot length semicolon. i plus, plus. Right parentheses. Line 126: left brace. Line 127: if left parentheses cell Bills left square bracket i right square bracket, exclamation mark, equals o b j C P dot cell Bills left square bracket i right square bracket Right parentheses. Line 128: return false semicolon. Line 129: right brace. Line 130: return true semicolon. Line 131: right brace. Line 132: right brace. Line 133: right brace.

Back to Example

this dot cell Bills equals cell Bills, semicolon, 2 forward slashes, incorrect, exclamation mark, Client still has reference, exclamation mark.

Back to Code

bills, left square bracket, 2, right square bracket, equals, 75.00, semicolon.

Back to Code

return cell Bills, semicolon, 2 forward slashes, incorrect, exclamation mark, Client has reference to instance variable.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, String, double, new, if, else. Line 1: Forward slash, asterisk. Asterisk. Client to exercise the Cell Phone class. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: import java dot text dot Decimal Format semicolon. Line 6: blank. Line 7: public class Cell Phone Client. Line 8: left brace. Line 9: public static void main, left parentheses, String left square bracket right square bracket a r g s, Right parentheses. Line 10: left brace. Line 11: double left square bracket right square bracket bills equals new double left square bracket 3 right square bracket semicolon. 2 forward slashes array of cell phone bills. Line 12: bills left square bracket 0 right square bracket equals 24.60 semicolon. 2 forward slashes assign values. Line 13: bills left square bracket 1 right square bracket equals 48.75 semicolon. Line 14: bills left square bracket 2 right square bracket equals 62.50 semicolon. Line 15: blank. Line 16: 2 forward slashes instantiate Cell Phone object using default constructor. Line 17: Cell Phone c 1 equals new Cell Phone left parentheses, right parentheses. semicolon. Line 18: blank. Line 19: 2 forward slashes instantiate two identical Cell Phone objects. Line 20: Cell Phone c 2 equals new Cell Phone, left parentheses double quotes 555 hyphen 555 hyphen 5555 double quotes , bills Right parentheses. semicolon. Line 21: Cell Phone c 3 equals new Cell Phone left parentheses double quotes 555 hyphen 555 hyphen5555 double quotes , bills Right parentheses. semicolon. Line 22: blank. Line 23: 2 forward slashes print data from c 1 and c 2. Line 24: System dot out dot print l n, left parentheses double quotes c 1 equals double quotes plus sign c 1 dot to String left parentheses, right parentheses. Right parentheses. semicolon. Line 25: System dot out dot print l n, left parentheses double quotes c 2 equals double quotes plus sign c 2 dot to String left parentheses, right parentheses. Right parentheses. semicolon. Line 26: blank. Line 27: 2 forward slashes find and print maximum bill. Line 28: Decimal Format money equals new Decimal Format, left parentheses double quotes dollar sign, 2 hashes, 0.00 double quotes Right parentheses.

semicolon. Line 29: System dot out dot print l n, left parentheses double quotes backslash n The highest bill is double quotes. Line 30: plus sign money dot format, left parentheses c 2 dot find Maximum Bill left parentheses, right parentheses. Right parentheses. Right parentheses. semicolon. Line 31: blank. Line 32: 2 forward slashes find and print total of all bills. Line 33: System dot out dot print l n, left parentheses double quotes backslash n The total of all bills is double quotes. Line 34: plus sign money dot format, left parentheses c 2 dot c a l c Total Bills left parentheses, right parentheses. Right parentheses. Right parentheses. semicolon. Line 35: blank. Line 36: System dot out dot print l n, left parentheses, right parentheses. semicolon. 2 forward slashes print blank line. Line 37: 2 forward slashes call equals method. Line 38: if left parentheses c 2 dot equals, left parentheses, c 3, Right parentheses. Right parentheses. Line 39: System dot out dot print l n, left parentheses double quotes c 2 and c 3 are equal double quotes Right parentheses. semicolon. Line 40: else. Line 41: System dot out dot print l n, left parentheses double quotes c 2 and c 3 are not equal double quotes Right parentheses. semicolon. Line 42: blank. Line 43: 2 forward slashes test encapsulation. Line 44: 2 forward slashes set new value in original array. Line 45: bills left square bracket 2 right square bracket equals 100.00 semicolon. Line 46: 2 forward slashes print c 2 to show value in object not changed. Line 47: System dot out dot print l n, left parentheses double quotes backslash n after client changes original array backslash n double quotes. Line 48: plus sign double quotes c 2 equals double quotes plus sign c 2 dot to String left parentheses, right parentheses. Right parentheses. semicolon. Line 49: blank. Line 50: 2 forward slashes test encapsulation further. Line 51: 2 forward slashes get array of cell bills and store in new array. Line 52: double left square bracket right square bracket bills Copy equals c 2 dot get Cell Bills left parentheses, right parentheses. semicolon. Line 53: blank. Line 54: bills Copy left square bracket 1 right square bracket equals 50.00 semicolon. 2 forward slashes change value of one element. Line 55: 2 forward slashes print c 2 to show value in object not changed. Line 56: System dot out dot print l n, left parentheses double quotes backslash n after client changes returned array backslash n double quotes. Line 57: plus sign double quotes c 2 equals double quotes plus sign c 2 dot to String left parentheses, right parentheses. Right parentheses. semicolon. Line 58: right brace. Line 59: right brace.

Back to Example

public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, for, int. Line 1: Forward slash, asterisk. Asterisk. Print Command Line arguments. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: public class Command Line Arguments. Line 5: left brace. Line 6: public static void main, left parentheses String left square bracket right square bracket a r g s Right parentheses. Line 7: left brace. Line 8: System dot out dot print l n, left parentheses double quotes The number of parameters is double quotes. Line 9: plus sign a r g s dot length Right parentheses. semicolon. Line 10: for left parentheses int i equals 0 semicolon. i left angled bracket a r g s dot length semicolon. i plus, plus. Right parentheses. Line 11: left brace. Line 12: System dot out dot print l n, left parentheses double quotes a r g s left square bracket double quotes plus sign i plus sign double quotes right square bracket : double quotes plus sign a r g s left square bracket i right square bracket Right parentheses. semicolon. Line 13: right brace. Line 14: right brace. Line 15: right brace.

Back to Example

The command is as follows: java Command Line Arguments input dot t x t, blank space, output dot t x t.

Back to Code

Program code. The code contains the following keywords: import, public, class, final, int, private, String, new, void, for, if, else, return. Line 1: Forward slash, asterisk.

Asterisk. Winners of Free Movies and Other Prizes. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: import java dot util dot Random semicolon. Line 6: blank. Line 7: public class Movie Winners. Line 8: left brace. Line 9: public final int MEMBERS equals 5000 semicolon. Line 10: 2 forward slashes array to hold winning member numbers chosen at random. Line 11: private int left square bracket right square bracket winners semicolon. Line 12: 2 forward slashes parallel array that holds prizes. Line 13: private String left square bracket right square bracket prizes equals left brace. double quotes 2 free movie tickets, exclamation mark, double quotes. Line 14: double quotes 1 free movie ticket, exclamation mark, double quotes. Line 15: double quotes free popcorn, exclamation mark, double quotes. Line 16: double quotes free box of candy, exclamation mark, double quotes right brace. semicolon. Line 17: Forward slash, asterisk. Asterisk. Default constructor instantiates winners array. Line 18: Asterisk. and randomly generates winning member I D s. Line 19: Asterisk, Forward slash. Line 20: public Movie Winners left parentheses, right parentheses. Line 21: left brace. Line 22: winners equals new int left square bracket prizes dot length right square bracket semicolon. Line 23: fill Winners left parentheses, right parentheses. semicolon. 2 forward slashes generate winner member I D s. Line 24: right brace. Line 25: blank. Line 26: Forward slash, asterisk. Asterisk. Utility method generates winner member I D s. Line 27: Asterisk. and stores them in the winners array. Line 28: Asterisk, Forward slash. Line 29: private void fill Winners left parentheses, right parentheses. Line 30: left brace. Line 31: Random rand equals new Random left parentheses, right parentheses. semicolon. Line 32: for left parentheses int i equals 0 semicolon. i left angled bracket winners dot length semicolon. i plus, plus. Right parentheses. Line 33: left brace. Line 34: winners left square bracket i right square bracket equals rand dot next Int, left parentheses MEMBERS Right parentheses. plus sign 1 semicolon. Line 35: right brace. Line 36: right brace. Line 37: blank. Line 38: Forward slash, asterisk. Asterisk. Calls index Of Winner with the member number. Line 39: Asterisk. then translates return value into the prize won. Line 40: Asterisk. at symbol param member, Number value to find. Line 41: Asterisk. at symbol return prize. Line 42: Asterisk, Forward slash. Line 43: public String get Prize, left parentheses int member Number Right parentheses. Line 44: left brace. Line 45: int prize Index equals index Of Winner, left parentheses member Number Right parentheses. semicolon. Line 46: if left parentheses prize Index equals equals negative 1 Right parentheses. Line 47: return double quotes Sorry, member is not a winner. double quotes semicolon. Line 48: else. Line 49: return double quotes You win double quotes plus sign prizes left square bracket prize Index right square bracket semicolon. Line 50: right brace. Line 51: blank. Line 52: Forward slash, asterisk. Asterisk. Performs sequential search of winners array. Line 53: Asterisk. at symbol param key member I D to find in winners array. Line 54: Asterisk. at symbol return index of key if found, negative 1 if not found. Line 55: Asterisk, Forward slash. Line 56: private int index Of Winner, left parentheses int key Right parentheses. Line 57: left brace. Line 58: for left parentheses int i equals 0 semicolon. i left angled bracket winners dot length semicolon. i plus, plus. Right parentheses. Line 59: left brace. Line 60: if left parentheses winners left square bracket i right square bracket equals equals key Right parentheses. Line 61: return i semicolon. Line 62: right brace. Line 63: return minus 1 semicolon. Line 64: right brace. Line 65: blank. Line 66: Forward slash, asterisk. Asterisk. Returns printable version of Movie Winners object. Line 67: Asterisk. at symbol return winning numbers separated by a tab. Line 68: Asterisk, Forward slash. Line 69: at symbol Override. Line 70: public String to String left parentheses, right parentheses. Line 71: left brace. Line 72: String return Value equals double quotes double quotes semicolon. Line 73: for left parentheses int i equals 0 semicolon. i left angled bracket winners dot length semicolon. i plus, plus. Right parentheses. Line 74: left brace. Line 75: return Value plus sign equals winners left square bracket i right square bracket plus sign double quotes backslash t double quotes semicolon. Line 76: right brace. Line 77: return return Value semicolon. Line 78: right brace. Line 79: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, new, while. Line 1: Forward slash, asterisk. Asterisk. Client for the Movie Winners class. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: import java dot util dot Scanner semicolon. Line 5: blank. Line 6: public class Movie Winners Client. Line 7: left brace. Line 8: public static void main, left parentheses String left square bracket right square bracket a r g s Right parentheses. Line 9: left brace. Line 10: 2 forward slashes instantiate the winning I D s array. Line 11: Movie Winners winning I D s equals new Movie Winners left parentheses, right parentheses. semicolon. Line 12: blank. Line 13: 2 forward slashes prompt for the member I D. Line 14: Scanner scan equals new Scanner, left parentheses System.in Right parentheses. semicolon. Line 15: System dot out dot print, left parentheses double quotes Enter the member's I D double quotes. Line 16: plus sign double quotes or 0 to stop, right angled bracket, double quotes Right parentheses. semicolon. Line 17: int search I D equals scan dot next Int left parentheses, right parentheses. semicolon. Line 18: blank. Line 19: while left parentheses search I D, exclamation mark, equals 0 Right parentheses. Line 20: left brace. Line 21: 2 forward slashes determine whether member is a winner. Line 22: System dot out dot print l n, left parentheses winning I D s dot get Prize, left parentheses, search I D, Right parentheses. Right parentheses. semicolon. Line 23: blank. Line 24: System dot out dot print, left parentheses double quotes backslash n Enter the next member's I D double quotes. Line 25: plus sign double quotes or 0 to stop, right angled bracket, double quotes Right parentheses. semicolon. Line 26: search I D equals scan dot next Int left parentheses, right parentheses. semicolon. Line 27: right brace. Line 28: blank. Line 29: System dot out dot print l n, left parentheses double quotes backslash n The winners were double quotes. Line 30: plus sign winning I D s dot to String left parentheses, right parentheses. Right parentheses. semicolon. Line 31: right brace. Line 32: right brace.

Back to Example

The pseudocode is as follows: To sort an array with n elements in ascending order: Step 1. Consider m elements as a subarray with m equals n elements. Step 2. Find the index of a largest value in this subarray. Step 3. Swap the values of the element with the largest value and the element in the last position in the subarray. Step 4. Consider a new subarray of m equals m minus 1 elements by eliminating the last element in the previous subarray. Step 5. Repeat steps 2 through 4 until m equals 1.

Back to Code

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 17. Index: 0. Column 2. Value: 26 in bold font. Index: 1.

Column 3. Value: 5. Index: 2. Column 4. Value: 2. Index: 3. All the 4 columns are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 17. Index: 0. Column 2. Value: 2. Index: 1. Column 3. Value: 5. Index: 2. Column 4. Value: 26. Index: 3. Columns 1 to 3 are shaded and the shaded columns correspond to the Unsorted subarray. Column 4 is unshaded and it is marked Sorted element.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 5. Index: 0. Column 2. Value: 2. Index: 1. Column 3. Value: 17. Index: 2. Column 4. Value: 26. Index: 3. Columns 1 and 2 are shaded and the shaded columns correspond to the Unsorted subarray. Columns 3 and 4 are unshaded and they are marked Sorted elements.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 2. Index: 0. Column 2. Value: 5. Index: 1. Column 3. Value: 17. Index: 2. Column 4. Value: 26. Index: 3. Column 1 is shaded and it is marked Unsorted subarray. Columns 2 to 4 are unshaded. Columns 1 to 4 are marked Sorted elements.

Back to Table

Line 1: To swap elements a and b, colon. Line 2: 1. Assign the value of element a to the temporary variable. Line 3: 2. Assign the value of element b to element a. Line 4: 3. Assign the value in the temporary variable to element b.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, int, for, private, return. Line 1: Forward slash, asterisk. Sort Utility Class. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Sorter. Line 6: left brace. Line 7: Forward slash, asterisk. Asterisk. Uses Selection Sort to sort. Line 8: Asterisk. an integer array in ascending order. Line 9: Asterisk. at symbol param array the array to sort. Line 10: Asterisk, Forward slash. Line 11: public static void selection Sort, left parentheses int left square bracket right square bracket array Right parentheses. Line 12: left brace. Line 13: int temp semicolon. 2 forward slashes temporary location for swap. Line 14: int max semicolon. 2 forward slashes index of maximum value in subarray. Line 15: blank. Line 16: for left parentheses int i equals 0 semicolon. i left angled bracket array dot length minus 1 semicolon. i plus, plus. Right parentheses. Line 17: left brace. Line 18: 2 forward slashes find index of largest value in subarray. Line 19: max equals index Of Largest Element, left parentheses, array, array dot length minus I, Right parentheses. semicolon. Line 20: blank. Line 21: 2 forward slashes swap array left square bracket max right square bracket and array left square bracket array dot length minus i minus 1 right square bracket. Line 22: temp equals array left square bracket max right square bracket semicolon. Line 23: array left square bracket max right square bracket equals array left square bracket array dot length minus i minus 1 right square bracket semicolon. Line 24: array left square bracket array dot length minus i minus 1 right square bracket equals temp semicolon. Line 25: right brace. Line 26: right brace. Line 27: blank. Line 28: Forward slash, asterisk. Asterisk. Finds index of largest element. Line 29: Asterisk. at symbol param size the size of the subarray. Line 30: Asterisk. at symbol param array the array to search. Line 31: Asterisk. at symbol return the index of the largest element in the subarray. Line 32: Asterisk, Forward slash. Line 33: private static int index Of Largest Element, left parentheses int left square bracket right square bracket array, int size Right parentheses. Line 34: left brace. Line 35: int index equals 0 semicolon. Line 36: for left parentheses int i equals 1 semicolon. i left angled bracket size semicolon. i plus, plus. Right parentheses. Line 37: left brace. Line 38: if left parentheses array left square bracket i right square bracket > array left square bracket index right square bracket Right parentheses. Line 39: index equals i semicolon. Line 40: right brace. Line 41: return index semicolon. Line 42: right brace. Line 43: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, int, new, for. Line 1: Forward slash, asterisk. Asterisk. Client for Selection Sort. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: import java dot util dot Random semicolon. Line 5: blank. Line 6: public class Selection Sort Client. Line 7: left brace. Line 8: public static void main, left parentheses String left square bracket right square bracket a r g s Right parentheses. Line 9: left brace. Line 10: 2 forward slashes instantiate an array and fill with random values. Line 11: int left square bracket right square bracket numbers equals new int left square bracket 6 right square bracket semicolon. Line 12: Random rand equals new Random left parentheses, right parentheses. semicolon. Line 13: for left parentheses int i equals 0 semicolon. i left angled bracket numbers dot length semicolon. i plus, plus. Right parentheses. Line 14: left brace. Line 15: numbers left square bracket i right square bracket equals rand dot next Int, left parentheses 5000 Right parentheses. plus sign 1 semicolon. Line 16: right brace. Line 17: blank. Line 18: System dot out dot print l n, left parentheses double quotes Before Selection Sort, the array is double quotes Right parentheses. semicolon. Line 19: for left parentheses int i equals 0 semicolon. i left angled bracket numbers dot length semicolon. i plus, plus. Right parentheses. Line 20: System dot out dot print, left parentheses numbers left square bracket i right square bracket plus sign double quotes backslash t double quotes Right parentheses. semicolon. Line 21: System dot out dot print l n, left parentheses, right parentheses. semicolon. Line 22: blank. Line 23: Sorter dot selection Sort, left parentheses numbers Right parentheses. semicolon. 2 forward slashes sort the array. Line 24: blank. Line 25: System dot out dot print l n, left parentheses, double quotes backslash n After Selection Sort, the array is double quotes, Right parentheses. semicolon. Line 26: for left parentheses int i equals 0 semicolon. i left angled bracket numbers dot length semicolon. i plus, plus. Right parentheses. Line 27: System dot out dot print, left parentheses numbers left square bracket i right square bracket plus sign double quotes backslash t double quotes Right parentheses. semicolon. Line 28: System dot out dot print l n, left parentheses, right parentheses. semicolon. Line 29: right brace. Line 30: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, int, for, private, if, while, return. Line 1: Forward slash, asterisk. Sort Utility Class. Line 2: Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Sorter. Line 6: left brace. Line 7: Forward slash, asterisk. Asterisk. Performs a Selection Sort on. Line 8: Asterisk. an integer array. Line 9: Asterisk. at symbol param the array to sort. Line 10: Asterisk, Forward slash. Line 11: public static void selection Sort, left parentheses int left square bracket right square bracket array Right parentheses. Line 12: left brace. Line 13: int temp semicolon. 2 forward slashes temporary location for swap. Line 14: int max semicolon. 2 forward slashes index of maximum value in subarray. Line 15: blank. Line 16: for left parentheses int i equals 0 semicolon. i left angled bracket array dot length minus 1 semicolon. i plus, plus. Right parentheses. Line 17: left brace. Line 18: 2 forward slashes find index of largest value in subarray. Line 19: max equals index Of Largest Element, left parentheses array, array dot length minus i Right parentheses. semicolon. Line 20: blank. Line 21: 2 forward slashes swap array left square bracket max right square bracket and array left square bracket array dot length minus i minus 1 right square bracket. Line 22: temp equals array left square bracket max right square bracket semicolon. Line 23: array left square bracket max right square bracket equals array left square bracket array dot length minus i minus 1 right square bracket semicolon. Line 24: array left square bracket array dot length minus i minus 1 right square bracket equals temp semicolon. Line 25: right brace. Line 26: right brace. Line 27: blank. Line 28: Forward slash, asterisk. Asterisk. Finds index of largest element. Line 29: Asterisk. at symbol param size the size of the subarray. Line 30: Asterisk. at symbol return the index of the largest element in the subarray.

Line 31: Asterisk, Forward slash. Line 32: private static int index Of Largest Element, left parentheses int left square bracket right square bracket array, int size Right parentheses. Line 33: left brace. Line 34: int index equals 0 semicolon. Line 35: for left parentheses int i equals 1 semicolon. i left angled bracket size semicolon. i plus, plus. Right parentheses. Line 36: left brace. Line 37: if left parentheses array left square bracket i right square bracket > array left square bracket index right square bracket Right parentheses. Line 38: index equals i semicolon. Line 39: right brace. Line 40: return index semicolon. Line 41: right brace. Line 42: blank. Line 43: Forward slash, asterisk. Asterisk. Performs an Insertion Sort on an integer array. Line 44: Asterisk. at symbol param array array to sort. Line 45: Asterisk, Forward slash. Line 46: public static void insertion Sort, left parentheses int left square bracket right square bracket array Right parentheses. Line 47: left brace. Line 48: int j, temp semicolon. Line 49: blank. Line 50: for left parentheses int i equals 1 semicolon. i left angled bracket array dot length semicolon. i plus, plus. Right parentheses. Line 51: left brace. Line 52: j equals i semicolon. Line 53: temp equals array left square bracket i right square bracket semicolon. Line 54: blank. Line 55: while left parentheses j exclamation mark equals 0, 2 ampersands, array left square bracket j minus 1 right square bracket, right angled bracket, temp Right parentheses. Line 56: left brace. Line 57: array left square bracket j right square bracket equals array left square bracket j minus 1 right square bracket semicolon. Line 58: j minus minus semicolon. Line 59: right brace. Line 60: blank. Line 61: array left square bracket j right square bracket equals temp semicolon. Line 62: right brace. Line 63: right brace. Line 64: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, int, new, for. Line 1: Forward slash, asterisk. Asterisk. Client for Insertion Sort. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: import java dot util dot Random semicolon. Line 5: blank. Line 6: public class Insertion Sort Client. Line 7: left brace. Line 8: public static void main, left parentheses String left square bracket right square bracket a r g s, Right parentheses. Line 9: left brace. Line 10: 2 forward slashes instantiate an array and fill with random values. Line 11: int left square bracket right square bracket numbers equals new int left square bracket 6 right square bracket semicolon. Line 12: Random rand equals new Random left parentheses, right parentheses. semicolon. Line 13: for left parentheses int i equals 0 semicolon. i left angled bracket numbers dot length semicolon. i plus, plus. Right parentheses. Line 14: left brace. Line 15: numbers left square bracket i right square bracket equals rand dot next Int, left parentheses 5000 Right parentheses. plus sign 1 semicolon. Line 16: right brace. Line 17: blank. Line 18: System dot out dot print l n, left parentheses, double quotes Before Insertion Sort, the array is double quotes Right parentheses. semicolon. Line 19: for left parentheses int i equals 0 semicolon. i left angled bracket numbers dot length semicolon. i plus, plus. Right parentheses. Line 20: System dot out dot print, left parentheses numbers left square bracket i right square bracket plus sign double quotes backslash t double quotes Right parentheses. semicolon. Line 21: System dot out dot print l n, left parentheses, right parentheses. semicolon. Line 22: blank. Line 23: Sorter dot insertion Sort, left parentheses numbers Right parentheses. semicolon. 2 forward slashes sort the array. Line 24: blank. Line 25: System dot out dot print l n, left parentheses, double quotes backslash n After Insertion Sort, the array is double quotes Right parentheses. semicolon. Line 26: for left parentheses int i equals 0 semicolon. i left angled bracket numbers dot length semicolon. i plus, plus. Right parentheses. Line 27: System dot out dot print, left parentheses, numbers left square bracket i right square bracket plus sign double quotes backslash t double quotes Right parentheses. semicolon. Line 28: System dot out dot print l n, left parentheses, right parentheses. semicolon. Line 29: right brace. Line 30: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, int, new, if, else, while, return. Line 1: Forward slash, asterisk. Asterisk.

Binary Search. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner semicolon. Line 6: blank. Line 7: public class Binary Searcher. Line 8: left brace. Line 9: public static void main, left parentheses String left square bracket right square bracket a r g s, Right parentheses. Line 10: left brace. Line 11: 2 forward slashes define an array sorted in ascending order. Line 12: int left square bracket right square bracket numbers equals left brace. 3, 6, 7, 8, 12, 15, 22, 36, 45. Line 13: 48, 51, 53, 64, 69, 72, 89, 95 right brace. semicolon. Line 14: blank. Line 15: Scanner scan equals new Scanner, left parentheses System.in Right parentheses. semicolon. Line 16: System dot out dot print, left parentheses double quotes Enter a value to search for, right angled bracket, double quotes Right parentheses. semicolon. Line 17: int key equals scan dot next Int left parentheses, right parentheses. semicolon. Line 18: blank. Line 19: int index equals binary Search, left parentheses numbers, key Right parentheses. semicolon. Line 20: if left parentheses index exclamation mark, equals negative 1 Right parentheses. Line 21: System dot out dot print l n, left parentheses key plus sign double quotes found at index double quotes plus sign index Right parentheses. semicolon. Line 22: else. Line 23: System dot out dot print l n, left parentheses, key plus sign double quotes not found double quotes Right parentheses. semicolon. Line 24: right brace. Line 25: blank. Line 26: public static int binary Search, left parentheses int left square bracket right square bracket a r r, int key Right parentheses. Line 27: left brace. Line 28: int start equals 0 semicolon. Line 29: int end equals a r r dot length minus 1 semicolon. Line 30: int middle semicolon. Line 31: blank. Line 32: while left parentheses end > equals start Right parentheses. Line 33: left brace. Line 34: middle equals left parentheses start plus sign end Right parentheses. Forward slash 2 semicolon. 2 forward slashes element in middle of array. Line 35: blank. Line 36: if left parentheses a r r left square bracket middle right square bracket equals equals key Right parentheses. Line 37: left brace. Line 38: return middle semicolon. 2 forward slashes key found at middle. Line 39: right brace. Line 40: else if left parentheses a r r left square bracket middle right square bracket, left angled bracket, key Right parentheses. Line 41: left brace. Line 42: end equals middle minus 1 semicolon. 2 forward slashes search left side of array. Line 43: right brace. Line 44: else. Line 45: left brace. Line 46: start equals middle plus sign 1 semicolon. 2 forward slashes search right side of array. Line 47: right brace. Line 48: right brace. Line 49: return minus 1 semicolon. Line 50: right brace. Line 51: right brace.

Back to Example

Line 1: 2 forward slashes 1. 5 asterisks student writes this method. Line 2: Forward slash, asterisk. Asterisk. Searches for key in integer array named a r r. Line 3: 2 forward slashes a r r is an instance variable of the class and has been. Line 4: 2 forward slashes instantiated and filled with random values. Line 5: 2 forward slashes at symbol param key value to search for. Line 6: 2 forward slashes at symbol return if key is found, the index of the first element. Line 7: 2 forward slashes in array whose value is key semicolon. if key is not found. Line 8: 2 forward slashes the method returns minus 1. Line 9: Asterisk, Forward slash. Line 10: public int sequential Search, left parentheses int key Right parentheses. Line 11: left brace. Line 12: 2 forward slashes Note: To animate the algorithm, put this method call as the. Line 13: 2 forward slashes first statement in your for loop. Line 14: 2 forward slashes animate, left parentheses i Right parentheses. semicolon. Line 15: 2 forward slashes where i is the index of the current array element. Line 16: return 0 semicolon. 2 forward slashes replace this statement with your return statement. Line 17: right brace. 2 forward slashes end of sequential Search. Line 18: 2 forward slashes 2. 5 asterisks student writes this method. Line 19: Forward slash, asterisk. Asterisk. Sorts a r r in ascending order using the bubble sort algorithm. Line 20: Asterisk, Forward slash. Line 21: public void bubble Sort left parentheses, right parentheses. Line 22: left brace. Line 23: 2 forward slashes Note: To animate the algorithm, put this method call as the. Line 24: 2 forward slashes last statement in your innermost for loop. Line 25: 2 forward slashes animate, left parentheses i, j Right parentheses. semicolon. Line 26: 2 forward slashes where i is the value of the outer loop counter. Line 27: 2 forward slashes and j is the value of the inner loop counter. Line 28: 2 forward slashes or the index of the current array element. Line 29: right brace. 2 forward slashes end of bubble Sort.

Back to Example

Program code. The code contains the following keywords: import, public, class, final, int, private, new, return. Line 1: Forward slash, asterisk. Asterisk. Die class. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: import java dot util dot Random semicolon. Line 5: blank. Line 6: public class Die. Line 7: left brace. Line 8: public final int SIDES equals 6 semicolon. Line 9: private Random rand semicolon. Line 10: blank. Line 11: Forward slash, asterisk. Asterisk. default constructor. Line 12: Asterisk. instantiates the Random object. Line 13: Asterisk, Forward slash. Line 14: public Die left parentheses, right parentheses. Line 15: left brace. Line 16: rand equals new Random left parentheses, right parentheses. semicolon. Line 17: right brace. Line 18: blank. Line 19: Forward slash, asterisk. Asterisk. rolls the die. Line 20: Asterisk. at symbol return the value of the roll. Line 21: Asterisk, Forward slash. Line 22: public int roll left parentheses, right parentheses. Line 23: left brace. Line 24: return rand dot next Int, left parentheses SIDES Right parentheses. plus sign 1 semicolon. Line 25: right brace. Line 26: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, final, int, for. Line 1: Forward slash, asterisk. Asterisk. Die Count Class. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Die Count. Line 6: left brace. Line 7: public static void main, left parentheses String left square bracket right square bracket a r g s Right parentheses. Line 8: left brace. Line 9: final int FACES equals 6, NUMBER underscore OF underscore ROLLS equals 500 semicolon. Line 10: blank. Line 11: 2 forward slashes instantiate the counter array. Line 12: 2 forward slashes which sets initial values to 0. Line 13: int left square bracket right square bracket roll Count equals new int left square bracket FACES right square bracket semicolon. Line 14: blank. Line 15: 2 forward slashes instantiate the Die. Line 16: Die d1 equals new Die left parentheses, right parentheses. semicolon. Line 17: blank. Line 18: 2 forward slashes roll the die 500 times. Line 19: for left parentheses int i equals 1 semicolon. i left angled bracket equals NUMBER underscore OF underscore ROLLS semicolon. i plus, plus. Right parentheses. Line 20: left brace. Line 21: int my Roll equals d 1 dot roll left parentheses, right parentheses. semicolon. Line 22: roll Count left square bracket my Roll minus 1 right square bracket plus, plus. semicolon. 2 forward slashes increment the counter for roll. Line 23: right brace. Line 24: blank. Line 25: 2 forward slashes print count for each roll. Line 26: System dot out dot print l n, left parentheses double quotes Roll backslash t Count double quotes Right parentheses. semicolon. Line 27: for left parentheses int i equals 0 semicolon. i left angled bracket roll Count dot length semicolon. i plus, plus. Right parentheses. Line 28: left brace. Line 29: System dot out dot print l n, left parentheses, left parentheses, i plus sign 1 Right parentheses. plus sign double quotes backslash t double quotes plus sign roll Count left square bracket i right square bracket Right parentheses. semicolon. Line 30: right brace. Line 31: right brace. Line 32: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, final, int, new, for. Line 1: Forward slash, asterisk. Asterisk. Die Count 2 Class. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Die Count 2. Line 6: left brace. Line 7: public static void main, left parentheses String left square bracket right square bracket a r g s Right parentheses. Line 8: left brace. Line 9: final int FACES equals 7, NUMBER underscore OF underscore ROLLS equals 500 semicolon. Line 10: blank. Line 11: 2 forward slashes instantiate the counter array. Line 12: 2 forward slashes which sets initial values to 0. Line 13: int left square bracket right square bracket roll Count equals new int left square bracket FACES right square bracket semicolon. Line 14: blank. Line 15: 2 forward slashes instantiate the Die. Line 16: Die d1 equals new Die left parentheses, right parentheses.

semicolon. Line 17: blank. Line 18: 2 forward slashes roll the die 500 times. Line 19: for left parentheses int i equals 1 semicolon. i left angled bracket equals NUMBER underscore OF underscore ROLLS semicolon. i plus, plus. Right parentheses. Line 20: left brace. Line 21: int my Roll equals d 1 dot roll left parentheses, right parentheses. semicolon. Line 22: roll Count left square bracket my Roll right square bracket plus, plus. semicolon. 2 forward slashes increment the counter for roll. Line 23: right brace. Line 24: blank. Line 25: 2 forward slashes print count for each roll. Line 26: System dot out dot print l n, left parentheses, double quotes Roll backslash t Count double quotes Right parentheses. semicolon. Line 27: for left parentheses int i equals 1 semicolon. i left angled bracket roll Count dot length semicolon. i plus, plus. Right parentheses. Line 28: left brace. Line 29: System dot out dot print l n, left parentheses i plus sign double quotes backslash t double quotes plus sign roll Count left square bracket i right square bracket Right parentheses. semicolon. Line 30: right brace. Line 31: right brace. Line 32: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, private, String, double, int, static, final, new, this, void, if, null, for, return. Line 1: Forward slash, asterisk. Asterisk. Seller class. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4 import java dot text dot Decimal Format semicolon.

Line 5: blank. Line 6: public class Seller. Line 7: left brace. Line 8: private String seller Name semicolon. Line 9: private double rating semicolon. Line 10: private int number Of Ratings semicolon. Line 11: private static final Decimal Format ONE underscore PLACE. Line 12: equals new Decimal Format, left parentheses, double quotes 0.0 double quotes Right parentheses. semicolon. Line 13: blank. Line 14: Forward slash, asterisk. Asterisk. Constructor. Line 15: Asterisk. at symbol param seller Name seller name. Line 16: Asterisk, Forward slash. Line 17: public Seller, left parentheses String seller Name Right parentheses. Line 18: left brace. Line 19: this dot seller Name equals seller Name semicolon. Line 20: right brace. Line 21: blank. Line 22: Forward slash, asterisk. Asterisk. Calc Rating method. Line 23: Asterisk. at symbol param new Ratings 0 to many new ratings. Line 24: Asterisk. updates rating and number Of Ratings instance variables. Line 25: Asterisk, Forward slash. Line 26: public void calc Rating left parentheses int newRatings Right parentheses. Line 27: left brace. Line 28: if left parentheses new Ratings exclamation mark equals null 2 ampersands new Ratings dot length greater than 0 Right parentheses. Line 29: left brace. Line 30: int total New Ratings equals 0 semicolon. Line 31: for left parentheses int i equals 0 semicolon. i left angled bracket new Ratings dot length semicolon. i plus, plus. Right parentheses. Line 32: total New Ratings plus sign equals new Ratings left square bracket i right square bracket semicolon. Line 33: blank. Line 34: rating equals left parentheses left parentheses rating Asterisk. Number Of Ratings Right parentheses. plus sign total New Ratings Right parentheses. Line 35: forward slash, left parentheses new Ratings dot length plus sign number Of Ratings Right parentheses. semicolon. Line 36: number Of Ratings plus sign equals new Ratings dot length semicolon. Line 37: right brace. Line 38: 2 forward slashes else, no new ratings, so no change to instance variables. Line 39: right brace. Line 40: blank. Line 41: Forward slash, asterisk. Asterisk. to String method. Line 42: Asterisk. at symbol return the seller name. Line 43: Asterisk. and rating left parentheses formatted to 1 decimal place Right parentheses. Line 44: Asterisk, Forward slash. Line 45: at symbol Override. Line 46: public String to String left parentheses, right parentheses. Line 47: left brace. Line 48: return seller Name plus sign double quotes : rating double quotes plus sign ONE underscore PLACE dot format, left parentheses rating Right parentheses. semicolon. Line 49: right brace. Line 50: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, new, int. Line 1: Forward slash, asterisk. Asterisk. Seller Ratings hyphen client for Seller. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Seller Ratings. Line 6: left brace. Line 7: public static void main, left parentheses String left square bracket right square bracket a r g s, Right parentheses. Line 8: left brace. Line 9: Seller seller equals new Seller, left parentheses double quotes Mary double quotes Right parentheses. semicolon. Line 10: Blank. Line 11: seller dot calc Rating, left parentheses 4 Right parentheses. semicolon. 2 forward slashes one value. Line 12: System dot out dot print l n, left parentheses seller Right parentheses. Semicolon. Line 13: blank. Line 14: seller dot calc Rating left parentheses 5, 5 Right parentheses. semicolon. 2 forward slashes two values. Line 15: System dot out dot print l n, left parentheses seller Right parentheses. semicolon. Line 16: blank. Line 17: int left square bracket right square bracket array Of Ratings equals left brace. 4, 4, 5 right brace. semicolon. Line 18: seller dot calc Rating, left parentheses, array Of Ratings Right parentheses. semicolon. 2 forward slashes array. Line 19: System dot out dot print l n, left parentheses, seller, Right parentheses. semicolon. Line 20: blank. Line 21: seller dot calc Rating left parentheses, right parentheses. semicolon. 2 forward slashes no values. Line 22: System dot out dot print l n, left parentheses, seller Right parentheses. semicolon. Line 23: right brace. Line 24: right brace.

Back to Example

The syntax is as follows. data type, left square bracket, right square bracket, left square bracket, right square bracket, array Name, semicolon.

Back to Code

The syntax is as follows. data type, left square bracket, right square bracket, left square bracket, right square bracket, left square bracket, right square bracket, array Name, semicolon.

Back to Code

double, left square bracket, right square bracket, left square bracket, right square bracket, daily T e m p s, semicolon.

Back to Code

data type array Name, left square bracket, right square bracket, left square bracket, right square bracket, semicolon.

Back to Code

char, left square bracket, right square bracket, left square bracket, right square bracket, quizzes, semicolon. 2 forward slashes, each element is a char.

Back to Code

Auto, left square bracket, right square bracket, left square bracket, right square bracket, cars, semicolon.

Back to Code

The syntax is as follows: data type, left square bracket, right square bracket, left square bracket, right square bracket, array Name 1, comma, array Name 2, semicolon.

Back to Code

int, left square bracket, right square bracket, left square bracket, right square bracket, brian, comma, jon, semicolon.

Back to Code

double, left square bracket, 7, right square bracket, left square bracket, 52, right square bracket, daily Temps, semicolon.

Back to Code

The syntax is as follows: array Name equals new data type, left square bracket, e x p 1, right square bracket, left square bracket, e x p 2, right square bracket, semicolon. The caption below the syntax reads: where e x p 1 and e x p 2 are expressions that evaluate to integers and specify, respectively, the number of rows and the number of columns in the array.

Back to Code

Line 1: daily Temps equals new double, left square bracket, 52, right square bracket, left square bracket, 7, right square bracket, semicolon. 2 forward slashes, daily Temps has 52 rows. Line 2: 2 forward slashes, and 7 columns, comma. Line 3: 2 forward slashes, for a total of 364 elements.

Back to Code

Line 1: int number Of Students equals 25, semicolon. Line 2: int number Of Quizzes equals 10, semicolon. Line 3: quizzes equals new char, left square bracket, number Of Students, right square bracket, left square bracket, number Of Quizzes, right square bracket, semicolon. Line 4: 2 forward slashes, quizzes has 25 rows and 10 columns. Line 5: 2 forward slashes, for a total of 250 elements. Line 6: cars equals new Auto, left square bracket, 5, right square bracket, left square bracket, 50, right square bracket, semicolon. Line 7: 2 forward slashes, cars has 5 rows and 50 columns. Line 8: 2 forward slashes, cars will store 250 Auto objects. Line 9: brian equals new int, left square bracket, 80, right square bracket, left square bracket, 20, right square bracket, semicolon. Line 10: 2 forward slashes, brian has 80 rows and 20 columns. Line 11: 2 forward slashes, there are 80 games per season. Line 12: 2 forward slashes, brian played baseball for 20 seasons. Line 13: jon equals new int, left square bracket, 80, right square bracket, left square bracket, 10, right square bracket, semicolon. Line 14: 2 forward slashes, jon has 80 rows and 10 columns. Line 15: 2 forward slashes, jon played baseball for 10 seasons.

Back to Code

The syntax is as follows: data type, left square bracket, right square bracket, left square bracket, right square bracket, array Name, equals, new, data type, left square bracket, e x p 1, right square bracket, left square bracket, e x p 2, right square bracket, semicolon. The caption below the syntax is as follows: where e x p 1 and e x p 2 are expressions that evaluate to integers and specify, respectively, the number of rows and columns in the array.

Back to Code

double, left square bracket, right square bracket, left square bracket, right square bracket, daily T e m p s, equals, new double, left square bracket, 52, right square bracket, left square bracket, 7, right square bracket, semicolon.

Back to Code

Line 1: double, left square bracket, right square bracket, left square bracket, right square bracket, daily T e m p s, semicolon. Line 2: daily T e m p s, equals, new double, left square bracket, 52, right square bracket, left square bracket, 7, right square bracket, semicolon.

Back to Code

char, left square bracket, right square bracket, left square bracket, right square bracket, quizzes, equals, new char, left square bracket, 25, right square bracket, left square bracket, 10, right square bracket, semicolon.

Back to Code

Line 1: char, left square bracket, right square bracket, left square bracket, right square bracket, quizzes, semicolon. Line 2: quizzes, equals, new char, left square bracket, 25, right square bracket, left square bracket, 10, right square bracket, semicolon.

Back to Code

The syntax is as follows: data type, left square bracket, right square bracket, left square bracket, right square bracket, array Name, equals, left brace, left brace, value 0 0, comma, value 0 1, comma, dot, dot, dot, right brace, comma, left brace, value 1 0, comma, value 1 1, comma, right brace, comma, dot, dot, dot, right brace, semicolon. The caption below the syntax is as follows: where value M N is an expression that evaluates to the data type of the array and is the value to assign to the element at row M and column N.

Back to Code

int, left square bracket, right square bracket, left square bracket, right square bracket, numbers List 1, equals, left brace, left brace, 0, comma, 5, comma, 10, right brace, comma, left brace, 0, comma, 3, comma, 6, comma, 9, right brace, right brace, semicolon.

Back to Code

Line 1: int, left square bracket, right square bracket, left square bracket, right square bracket, grades, equals, new int, left square bracket, 2, right square bracket, left square bracket, 3, right square bracket, semicolon. Line 2: grades equals, left brace, left brace, 89, comma, 73, comma, 98, right brace, comma. Line 3: left brace, 88, comma, 65, comma, 92, right brace, right brace, semicolon.

Back to Code

Line 1: Auto sports Car, equals, new Auto, left parentheses, double quotes, Ferrari, double quotes, comma, 0, comma, 0.0, right parentheses, semicolon. Line 2: Auto sedan 1, equals, new Auto, left parentheses, double quotes, B M W, double quotes, 0, comma, 0.0, right parentheses, semicolon. Line 3: Auto sedan 2, equals, new Auto, left parentheses, double quotes, B M W, double quotes, comma, 100, comma, 15.0, right parentheses, semicolon. Line 4: Auto sedan 3, equals, new Auto, left parentheses, double quotes, Toyota, double quotes, comma, 0, comma, 0.0, right parentheses, semicolon. Line 5: Auto r v 1, equals, new Auto, left parentheses, double quotes, Jeep, double quotes, comma, 0, comma, 0.0, right parentheses, semicolon. Line 6: Auto, left square bracket, right square bracket, left square bracket, right square bracket, cars, equals, left brace, left brace, sports Car, comma, sedan 1, right brace, comma. Line 7: left brace, r v 1, comma, new Auto, left parentheses, right parentheses, right brace, comma. Line 8: left brace, sedan 2, comma, sedan 3, right brace, right brace, semicolon.

Back to Code

Line 1: char, left square bracket, right square bracket, left square bracket, right square bracket, grades, semicolon. 2 forward slashes, declare the array. Line 2: grades, equals, new char, left square bracket, 4, right square bracket, left square bracket, right square bracket, semicolon. 2 forward slashes, instantiate the array. Line 3: 2 forward slashes, grades has 4 null array elements.

Back to Code

Line 4: grades, left square bracket, 0, right square bracket, equals, new char, left square bracket, 23, right square bracket, semicolon. 2 forward slashes, instantiate row 0, semicolon, 23 char elements. Line 5: grades, left square bracket, 1, right square bracket, equals, new char, left square bracket, 16, right square bracket, semicolon. 2 forward slashes, instantiate row 1, semicolon, 16 char elements. Line 6: grades, left square bracket, 2, right square bracket, equals, new char, left square bracket, 12, right square bracket, semicolon. 2 forward slashes, instantiate row 2, semicolon, 12 char elements. Line 7: grades, left square bracket, 3, right square bracket, equals, new char, left square bracket, 28, right square bracket, semicolon. 2 forward slashes, instantiate row 3, semicolon, 28 char elements.

Back to Code

grades, equals, new char, left square bracket, 4, right square bracket, left square bracket, right square bracket, semicolon.

Back to Code

The syntax is as follows: array Name, left square bracket, e x p 1, right square bracket, left square bracket, e x p 2, right square bracket. The caption below the syntax is as follows: where e x p 1 and e x p 2 are expressions that evaluate to integers.

Back to Code

The syntax is as follows: array Name dot length.

Back to Code

The syntax is as follows: array Name, left square bracket, i, right square bracket, dot, length.

Back to Code

The table has 2 columns labeled Array Element and Syntax. The row entries are as follows. Row 1. Array Element: Row 0, column j. Syntax: array Name, left square bracket, 0, right square bracket, left square bracket, j, right square bracket. Row 2. Array Element: Row i, column j. Syntax: array Name, left square bracket, i, right square bracket, left square bracket, j, right square bracket. Row 3. Array Element: Last row, column j. Syntax: array Name, left square bracket, array Name, dot, length, minus 1, right square bracket, left square bracket, j, right square bracket. Row 4. Array Element: Last row, last column. Syntax: array Name, left square bracket, array Name dot length minus 1, right square bracket, left square bracket, array Name, left square bracket, array Name dot length minus 1, right square bracket, dot, length minus 1, right square bracket. Row 5. Array Element: Number of rows in the array. Syntax: array Name dot length. Row 6. Array Element: Number of columns in row i. Syntax: array Name, left square bracket, i, right square bracket, dot, length.

Back to Table

The table has 3 rows and 4 columns. The rows are labeled July, August, and September from top to bottom. The columns are labeled Joe, Jane, Mike, and Sarah from left to right. The row entries are as follows. Row 1. July. Joe: 45.24. Jane: 54.67. Mike: 32.55. Sarah: 25.61. Row 2. August. Joe: 65.29. Jane: 49.75. Mike: 32.08. Sarah: 26.11. Row 3. September. Joe: 75.24. Jane: 54.53. Mike: 34.55. Sarah: 28.16.

Back to Table

Program code. The code contains the following keywords: public, class, static, void, String, final, int, double, new, for. Line 1: Forward slash. Asterisk. Two hyphen Dimensional array of Cell Phone Bills. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk.

Forward slash. Line 4: blank. Line 5: public class Family Cell Bills. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: 2 forward slashes, declare constants for the number of rows and columns. Line 10, indented once: final int NUMBER underscore OF underscore MONTHS, equals, 3, semicolon. Line 11, indented once: final int NUMBER underscore OF underscore PERSONS, equals, 4, semicolon. Line 12: blank. Line 13, indented once: 2 forward slashes, declare and instantiate the array. Line 14, indented once: double, left square bracket, right square bracket, left square bracket, right square bracket, family Cell Bills, equals. Line 15, indented twice: new double, left square bracket, NUMBER underscore OF underscore MONTHS, right square bracket, left square bracket, NUMBER underscore OF underscore PERSONS, right square bracket, semicolon. Line 16: blank. Line 17, indented once: 2 forward slashes. assign values to array elements. Line 18, indented once: family Cell Bills, left square bracket, 0, right square bracket, left square bracket, 0, right square bracket, equals, 45.24, semicolon. 2 forward slashes, row 0. Line 19, indented once: family Cell Bills, left square bracket, 0, right square bracket, left square bracket, 1, right square bracket, equals, 54.67, semicolon. Line 20, indented once: family Cell Bills, left square bracket, 0, right square bracket, left square bracket, 2, right square bracket, equals, 32.55, semicolon.Line 21, indented once: family Cell Bills, left square bracket, 0, right square bracket, left square bracket, 3, right square bracket, equals, 25.61, semicolon. Line 22: blank. Line 23, indented once: family Cell Bills, left square bracket, 1, right square bracket, left square bracket, 0, right square bracket, equals, 65.29, semicolon. 2 forward slashes, row 1. Line 24, indented once: family Cell Bills, left square bracket, 1, right square bracket, left square bracket, 1, right square bracket, equals, 49.75, semicolon. Line 25, indented once: family Cell Bills, left square bracket, 1, right square bracket, left square bracket, 2, right square bracket, equals, 32.08, semicolon. Line 26, indented once: family Cell Bills, left square bracket, 1, right square bracket, left square bracket, 3, right square bracket, equals, 26.11, semicolon. Line 27: blank. Line 28, indented once: family Cell Bills, left square bracket, 2, right square bracket, left square bracket, 0, right square bracket, equals, 75.24, semicolon. 2 forward slashes, row 2. Line 29, indented once: family Cell Bills, left square bracket, 2, right square bracket, left square bracket, 1, right square bracket, equals, 54.53, semicolon. Line 30, indented once: family Cell Bills, left square bracket, 2, right square bracket, left square bracket, 2, right square bracket, equals, 34.55, semicolon. Line 31, indented once: family Cell Bills, left square bracket, 2, right square bracket, left square bracket, 3, right square bracket, equals, 28.16, semicolon. Line 32: blank. Line 33, indented once: System dot out dot print l n, left parentheses, double quotes, The first monthly cell bill for the first, double quotes. Line 34, indented twice: plus, double quotes, family member is, backslash, n, double quotes. Line 35, indented twice: plus, family Cell Bills, left square bracket, 0, right square bracket, left square bracket, 0, right square bracket, right parentheses, semicolon. Line 36, indented once: System dot out dot print l n, left parentheses, double quotes, The last monthly cell bill for the last, blank space, double quotes. Line 37, indented twice: plus, double quotes, family member is, backslash, n, double quotes. Line 38, indented twice: plus, family Cell Bills, left square bracket, NUMBER underscore OF underscore MONTHS minus 1, right square bracket, left square bracket, NUMBER underscore OF underscore PERSONS minus 1, right square bracket, right parentheses, semicolon. Line 39: blank. Line 40, indented once: int num Rows, equals, family Cell Bills dot length, semicolon. Line 41, indented once: System dot out dot print l n, left parentheses, double quotes, backslash n, The number of rows is, double quotes, plus, num Rows, right parentheses, semicolon. Line 42: blank. Line 43, indented once: for, left parentheses, int i equals 0, semicolon, i less than num Rows, semicolon, i plus, plus, right parentheses. Line 44, indented once: left brace. Line 45, indented twice: System dot out dot print, left parentheses, double quotes, The number of columns in row, double quotes, plus, i, plus, double quotes, blank space, is, double quotes, right parentheses, semicolon. Line 46, indented twice: System dot out dot print l n, left parentheses, family Cell Bills, left square bracket, i, right square bracket, dot, length, right parentheses, semicolon. Line 47, indented twice: right brace. Line 48, indented once: right brace. Line 49: right brace.

Back to Example

a r r, left square bracket, a r r dot length, minus, 1, right square bracket, left square bracket, a r r, left square bracket, a r r dot length minus 1, right square bracket, dot length minus 1, right square bracket.

Back to Code

Line 1: 2 forward slashes. invalid row indexes for the family cell Bills array, 2 exclamation marks. Line 2: family Cell Bills, left square bracket, negative 1, right square bracket, left square bracket, 2, right square bracket. Line 3: 2 forward slashes. the lowest valid row index is 0. Line 4: family Cell Bills, left square bracket, cell Bills dot length, right square bracket, left square bracket, 2, right square bracket. Line 5: 2 forward slashes. the highest valid row index is family Cell Bills dot length minus 1.

Back to Code

Line 1: 2 forward slashes. invalid column indexes for the family cell Bills array, 2 exclamation marks. Line 2: family Cell Bills, left square bracket, 1, right square bracket, left square bracket, negative 1, right square bracket. Line 3: 2 forward slashes. the lowest valid column index is 0. Line 4: family Cell Bills, left square bracket, 1, right square bracket, left square bracket, family Cell Bills, left square bracket, 1, right square bracket, dot, length, right square bracket. Line 5: 2 forward slashes. the highest valid column index of row i is. Line 6: 2 forward slashes. family Cell Bills, left square bracket, i, right square bracket, dot, length minus 1.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, new, Auto. Line 1: Forward slash. Asterisk. Working with a two hyphen dimensional array of objects. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Two D i m Auto Array. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: 2 forward slashes. instantiate several Auto object references. Line 10, indented twice: Auto sedan 1, equals, new Auto, left parentheses, double quotes, B M W, double quotes, 0, comma, 0.0, right parentheses, semicolon. Line 11, indented twice: Auto sedan 2, equals, new Auto, left parentheses, double quotes, B M W, double quotes, comma, 100, comma, 15.0, right parentheses, semicolon. Line 12, indented twice: Auto sedan 3, equals, new Auto, left parentheses, double quotes, Toyota, double quotes, comma, 0, comma, 0.0, right parentheses, semicolon. Line 13, indented twice:

Auto sports Car, equals, new Auto, left parentheses, double quotes, Ferrari, double quotes, comma, 0, comma, 0.0, right parentheses, semicolon. Line 14, indented twice: Auto r v 1, equals, new Auto, left parentheses, double quotes, Jeep, double quotes, comma, 0, comma, 0.0, right parentheses, semicolon. Line 15, indented twice: Auto r v 2, equals, new Auto, left parentheses, double quotes, Ford, double quotes, comma, 200, comma, 30.0, right parentheses, semicolon. Line 16: blank. Line 17, indented twice: 2 forward slashes. declare and initialize two hyphen dimensional array of Autos. Line 18, indented twice: Auto, left square bracket, right square bracket, left square bracket, right square bracket, cars, equals, left brace, left brace, sedan 1, comma, sedan 2, comma, sedan 3, right parentheses, comma. Line 19, indented multiple times: left brace, sports Car, comma, new Auto, left parentheses, right parentheses, right brace, comma. Line 20, indented multiple times: left brace, r v 1, comma, r v 2, right parentheses, right parentheses, semicolon. Line 21: blank. Line 22, indented twice: Auto retrieved Car, equals, cars, left square bracket, 1, right square bracket, left square bracket, 0, right square bracket, semicolon. Line 23, indented twice: 2 forward slashes. retrieved Car gets the sports Car object reference. Line 24: blank. Line 25, indented twice: System dot out dot print l n, left parentheses, double quotes, cars, left square bracket, 1, right square bracket, left square bracket, 0, right square bracket, apostrophe, s, description is, colon, backslash, n, double quotes. Line

26, indented multiple times: plus, retrieved Car, right parentheses, semicolon. Line 27, indented once: right brace. Line 28: right brace.

Back to Example

for, left parentheses, int i equals 0, semicolon, i less than array Name dot length, semicolon, i plus, plus, right parentheses.

Back to Code

int i equals 0, semicolon.

Back to Code

i less than array Name dot length.

Back to Code

for, left parentheses, int j equals 0, semicolon, j less than array Name, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses.

Back to Code

int j equals 0, semicolon.

Back to Code

j less than array Name, left square bracket, i, right square bracket, dot, length.

Back to Code

array Name, left square bracket, i, right square bracket, left square bracket, j, right square bracket.

Back to Code

Program code. The code contains the following keywords: for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i less than array Name dot length, semicolon, i plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int j equals 0, semicolon, j less than array Name, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: 2 forward slashes. process element array Name, left square bracket, i, right square bracket, left square bracket, j, right square bracket. Line 6, indented once: right brace. Line 7: right brace.

Back to Code

i less than or equal to array Name dot length.

Back to Code

j less than or equal to array Name, left square bracket, i, right square bracket, dot, length.

Back to Code

array Name, left square bracket, i, right square bracket, dot, length minus 1.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, double, for, int. Line 1: Forward slash. Asterisk. Processing a two hyphen dimensional array of Cell Phone Bills. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Output Family Cell Bills. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented once: 2 forward slashes. declare and initialize the array. Line 10, indented once: double, left square bracket, right square bracket, left square bracket, right square bracket, family Cell Bills, equals, left brace, left brace, 45.24, comma, 54.67, comma, 32.55, comma, 25.61, right brace, comma. Line 11, indented multiple times: left brace, 65.29, comma, 49.75, comma, 32.08, comma, 26.11, right brace, comma. Line 12, indented multiple times: left brace, 75.24, comma, 54.53, comma, 34.55, comma, 28.16, right brace, right brace, semicolon. Line 13: blank. Line 14, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, t, Data for family cell bills, double quotes, right parentheses, semicolon. Line 15: blank. Line 16, indented twice: for, left parentheses, int i, equals, 0, semicolon, i less than family Cell Bills dot length, semicolon, i plus, plus, right parentheses. Line 17, indented twice: left brace. Line 18, indented thrice: System dot out dot print, left parentheses, double quotes, backslash, n, row, blank space, double quotes, plus, i, plus, double quotes, colon, backslash, t, double quotes, right parentheses, semicolon. Line 19, indented thrice: for, left parentheses, int j equals 0, semicolon, j less than family Cell Bills, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 20, indented thrice: left brace. Line 21, indented 4 times: System dot out dot print, left parentheses, family Cell Bills, left square bracket, i, right square bracket, left square bracket, j, right square bracket, plus, double quotes, backslash, t, double quotes, right parentheses, semicolon. Line 22, indented thrice: right brace. Line 23, indented twice: right brace. Line 24, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 25, indented once: right brace. Line 26: right brace.

Back to Example

Program code. The code contains the following keywords: for, int. Line 1: for, left parentheses, int j equals 0, semicolon, j less than array Name, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes. process element array Name, left square bracket, i, right square bracket, left square bracket, j, right square bracket. Line 4: right brace.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, String, double, for, int, new, do, while, for. Line 1: Forward slash, asterisk. Processing One Row of a Two hyphen Dimensional Array. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: import java dot text dot Number Format, semicolon. Line 7: blank. Line 8: public class Sum A Row Family Cell Bills. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: 2 forward slashes, declare and initialize the array. Line 13, indented twice: double, left square bracket, right square bracket, left square bracket, right square bracket, family Cell Bills, equals, left brace, left brace, 45.24, comma, 54.67, comma, 32.55, comma, 25.61, right brace, comma. Line 14, indented multiple times: left brace, 65.29, comma, 49.75, comma, 32.08, comma, 26.11, right brace, comma. Lien 15, indented multiple times: left brace, 75.24, comma, 54.53, comma, 34.55, comma, 28.16, right brace, right brace, semicolon. Line 16: blank. Line 17, indented twice: String, left square bracket, right square bracket, months, equals, left brace, double quotes, July, double quotes, comma, double quotes, August, double quotes, comma, double quotes, September, double quotes, right brace, semicolon. Line 18, indented twice: for, left parentheses, int i equals 0, semicolon, i less than months dot length, semicolon, i plus, plus, right parentheses. Line 19, indented thrice: System dot out dot print l n, left parentheses, double quotes, Month, blank space, double quotes, plus, i, plus, double quotes, blank space, colon, blank space, double quotes, plus, months, left square bracket, i, right square bracket, right parentheses, semicolon. Line 20: blank. Line 21, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 22, indented twice: int current Month, semicolon. Line 23, indented twice: do. Line 24, indented twice: left brace. Line 25, indented thrice: System dot out dot print, left parentheses, double quotes, Enter a month number between 0 and 2, right angled bracket, blank space, double quotes, right parentheses, semicolon. Line 26, indented thrice: current Month, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 27, indented twice: right brace, while, left parentheses, current Month less than 0, 2 pipes, current Month greater than 2, right parentheses, semicolon. Line 28: blank. Line 29, indented twice: double monthly Family Bills, equals, 0.0, semicolon. Line 30, indented twice: for, left parentheses, int j equals 0, semicolon, j less than family Cell Bills, left square bracket, current Month, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 31, indented twice: left brace. Line 32, indented thrice: 2 forward slashes, add current family member bill to total. Line 33, indented thrice: monthly Family Bills, plus, equals, family Cell Bills, left square bracket, current Month, right square bracket, left square bracket, j, right square bracket, semicolon. Line 34, indented twice: right brace. Line 35: blank. Line 36, indented twice: Number Format price Format, equals, Number Format dot get Currency Instance, left parentheses, right parentheses, semicolon. Line 37, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash n, The total family cell bills during, blank space, double quotes. Line 38, indented multiple times: plus, months, left square bracket, current Month, right square bracket, plus, double quotes, blank space, is, blank space, double quotes. Line 39, indented multiple times: plus, price Format dot format, left parentheses, monthly Family Bills, right parentheses, right parentheses, semicolon. Line 40, indented once: right brace. Line 41: right brace.

Back to Example

Program code. The code contains the following keywords: for, int, if. Line 1: for, left parentheses, int i equals 0, semicolon, i less than array Name dot length, semicolon, i, plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, j less than array Name, left square bracket, i, right square bracket, dot, length, right parentheses. Line 4, indented twice: 2 forward slashes, process element array Name, left square bracket, i, right square bracket, left square bracket, j, right square bracket.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, String, double, for, int, do, while, for, if. Line 1: Forward slash. Asterisk. Processing One Column of a Two hyphen Dimensional Array. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: import java dot text dot Number Format, semicolon. Line 7: blank. Line 8: public class Max Member Bill. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: 2 forward slashes. declare and initialize the array. Line 13, indented twice: double, left square bracket, right square bracket, left square bracket, right square bracket, family Cell Bills, equals, left brace, left brace, 45.24, comma, 54.67, comma, 32.55, comma, 25.61, right brace, comma. Line 14, indented multiple times: left brace, 65.29, comma, 49.75, comma, 32.08, comma, 26.11, right brace. Line 15, indented multiple times: left brace, 75.24, comma, 54.53, comma, 34.55, comma, 28.16, right brace, right brace, semicolon. Line 16: blank. Line 17, indented once: String, left square bracket, right square bracket, family Members, equals, left brace, double quotes, Joe, double quotes, comma, double quotes, Jane, double quotes, comma, double quotes, Mike, double quotes, comma, double quotes, Sarah, double quotes, right brace, semicolon. Line 18, indented twice: for, left parentheses, int i equals 0, semicolon, i less than family Members dot length, semicolon, i, plus, plus, right parentheses. Line 19, indented thrice: System dot out dot print l n, left parentheses, double quotes, Family member, blank space, double quotes, plus, i, plus, double quotes, blank space, colon, blank space, double quotes. Line 20, indented multiple times, plus, family Members, left square bracket, i, right square bracket, right parentheses, semicolon. Line 21: blank. Line 22, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot int, right parentheses, semicolon. Line 23, indented twice: int current Member, semicolon. Line 24, indented twice: do. Line 25, indented twice: left brace. Line 26, indented thrice: System dot out dot print, left parentheses, double quotes, Enter a family member between 0 and 3, right angled bracket, double quotes, right parentheses, semicolon. Line 27, indented thrice: current Member equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 28, indented twice: right brace, while, left parentheses, current Member less than 0, 2 pipes, current Member greater than 3, right parentheses, semicolon. Line 29: blank. Line 30, indented twice: double member Max Bill, equals, family Cell Bills, left square bracket, 0, right square bracket, left square bracket, current Member, right square bracket, semicolon. Line 31, indented twice: for, left parentheses, int i equals 1, semicolon, i less than family Cell Bills dot length, semicolon, i, plus, plus, right parentheses. Line 32, indented twice: left brace. Line 33, indented thrice: if, left parentheses, current Member less than family Cell Bills, left square bracket, i, right square bracket, dot length, right parentheses. Line 34, indented thrice: left brace. Line 35, indented 4 times: 2 forward slashes, update member Max Bill if necessary. Line 36, indented 4 times: if, left parentheses, family Cell Bills, left square bracket, i, right square bracket, left square bracket, current Member, right square bracket, greater than, member Max Bill, right parentheses. Line 37, indented 5 times: member Max Bill, equals, family Cell Bills, left square bracket, i, right square bracket, left square bracket, current Member, right square bracket, semicolon. Line 38, indented thrice: right brace. Line 39, indented twice: right brace. Line 40: blank. Line 41, indented twice: Number Format price Format, equals, Number Format dot get Currency Instance, left parentheses, right parentheses, semicolon. Line 42, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, The max cell bill for, blank space, double quotes. Line 43, indented multiple times: plus, family Members, left square bracket, current Member, right square bracket, plus, double quotes, blank space, is, blank space, double quotes. Line 44, indented multiple times: plus, price Format dot format, left parentheses, member Max Bill, right parentheses, right parentheses, semicolon. Line 45, indented once:

right brace. Line 46: right brace.

Back to Example

Program code. The code contains the following keywords: for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i less than array Name dot length, semicolon, i plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, initialize processing variables for row i. Line 4, indented once: for, left parentheses, int j equals 0, semicolon, j less than array Name, left square bracket, i, right square bracket, dot length, semicolon, j plus, plus, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: 2 forward slashes, process element array Name, left square bracket, i, right square bracket, left square bracket, j, right square bracket. Line 7, indented once: right brace. Line 8, indented once: 2 forward slashes, finish the processing of row i. Line 9: right brace.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, String, double, for, int. Line 1: Forward slash. Asterisk. Processing Each Row of a Two hyphen Dimensional Array. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: import java dot text dot Number Format, semicolon. Line 7: blank. Line 8: public class Sum Each Row Family Cell Bills. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: 2 forward slashes, declare and initialize the array. Line 13, indented twice: double, left square bracket, right square bracket, left square bracket, right square bracket, family Cell Bills, equals, left brace, left brace, 45.24, comma, 54.67, comma, 32.55, comma, 25.61, right brace, comma. Line 14, indented multiple times: left brace, 65.29, comma, 49.75, comma, 32.08, comma, 26.11, right brace, comma. Line 15, indented multiple times: left brace, 75.24, comma, 54.53, comma, 34.55, comma, 28.16, right brace, right brace, semicolon. Line 16: blank. Line 17, indented twice: String, left square bracket, right square bracket, months, equals, left brace, double quotes, July, double quotes, comma, double quotes, August, double quotes, comma, double quotes, September, double quotes, right brace, semicolon. Line 18: blank. Line 19, indented twice: Number Format price Format, equals, Number Format dot get Currency Instance, left parentheses, right parentheses, semicolon. Line 20, indented twice: double current Month Total, semicolon. Line 21, indented twice: for, left parentheses, int i equals 0, semicolon, i less than family Cell Bills dot length, semicolon, i plus, plus, right parentheses. Line 22, indented twice: left brace. Line 23, indented thrice: current Month Total, equals, 0.0, semicolon. 2 forward slashes, initialize total for row. Line

24, indented thrice: for, left parentheses, int j equals 0, semicolon, j less than family Cell Bills, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 25, indented thrice: left brace. Line 26, indented 4 times: 2 forward slashes. add current family member bill to current monthly total. Line 27, indented 4 times: current Month Total, plus, equals, family Cell Bills, left square bracket, i, right square bracket, left square bracket, j, right square bracket, semicolon. Line 28, indented thrice: right brace. Line 29, indented thrice: 2 forward slashes, print total for row. Line 30, indented thrice: System dot out dot print l n, left parentheses, double quotes, The total for, blank space, double quotes, plus, months, left square bracket, i, right square bracket, plus, double quotes, blank space, is, blank space, double quotes. Line 31, indented multiple times: plus, price Format dot format, left parentheses, current Month Total, right parentheses, right parentheses, semicolon. Line 32, indented twice: right brace. Line 33, indented once: right brace. Line 34: right brace.

Back to Example

Line 1: int, left square bracket, right square bracket, left square bracket, right square bracket, grades, equals, left brace, left brace, 89, comma, 75, right brace, comma. Line 2, indented multiple times: left brace, 84, comma, 76, comma, 92, comma, 96, right brace, comma. Line 3, indented multiple times: left brace, 80, comma, 88, comma, 95, right brace, right brace, semicolon.

Back to Code

Program code. The code contains the following keywords: for, int, if. Line 1: 2 forward slashes, max Number Of Columns holds the number of columns. Line 2: 2 forward slashes, in the largest row of family Cell Bills. Line 3: for, left parentheses, int j equals 0, semicolon, j less than max Number Of Columns, semicolon, j plus plus, right parentheses. Line 4: left brace. Line 5, indented once: for, left parentheses, int i equals 0, semicolon, i less than array Name dot length, semicolon, i plus, plus, right parentheses. Line 6, indented once: left brace. Line 7, indented twice: if, left parentheses, j less than array Name, left square bracket, i, right square bracket, dot, length, right parentheses. Line 8, indented twice: left brace. Line 9, indented thrice: 2 forward slashes. process element array Name, left square bracket, i, right square bracket, left square bracket, j, right square bracket. Line 10, indented twice: right brace. Line 11, indented once: right brace. Line 12: right brace.

Back to Code

j less than max Number Of Columns.

Back to Code

i less than array Name dot length.

Back to Code

if, left parentheses, j less than array Name, left square bracket, i, right square bracket, dot, length, right parentheses.

Back to Code

Program code. The code contains the following keywords: public, class, static, void, String, int, for, if. Line 1: Forward slash. Asterisk. Processing Each Column in a Two hyphen Dimensional Array. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Grades Processing. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented once: int, left square bracket, right square bracket, left square bracket, right square bracket, grades, equals, left brace, left brace, 89, comma, 75, right brace, comma. Line 10, indented multiple times: left brace, 84, comma, 76, comma, 92, comma, 96, right brace, comma. Line 11, indented multiple times: left brace, 80, comma, 88, comma, 95, right brace, right brace, semicolon. Line 12: blank. Line 13, indented once: 2 forward slashes. compute the maximum number of columns. Line 14, indented once: int max Number Of Columns, equals, grades, left square bracket, 0, right square bracket, dot, length, semicolon. Line 15, indented once: for, left parentheses, int i equals 1, semicolon, i less than grades, left square bracket, 0, right square bracket, dot, length, semicolon. Line 15, indented once: for, left parentheses, int i equals 1, semicolon, i less than grades dot length, semicolon, i plus, plus, right parentheses. Line 16, indented once: left brace. Line 17, indented twice: if, left parentheses, grades, left square bracket, i, right square bracket, dot, length, greater than, max Number Of Columns, right parentheses. Line 18, indented thrice: max Number Of Columns, equals, grades, left square bracket, i, right square bracket, dot, length, semicolon. Line 19, indented once: right brace. Line 20, indented once: System dot out dot print l n, left parentheses, double quotes, The maximum number of columns in grades is, blank space, double quotes. Line 21, indented multiple times: plus, max Number Of Columns, right parentheses, semicolon. Line 22: blank. Line 23, indented once: for, left parentheses, int j equals 0, semicolon, j less than max Number Of Columns, semicolon, j plus, plus, right parentheses. Line 24, indented once: left brace. Line 25, indented twice: System dot out dot print, left parentheses, double quotes, backslash, n, Column, blank space, double quotes, plus, j, plus, double, colon, blank space, double quotes, right parentheses, semicolon. Line 26, indented twice: for, left parentheses, int i equals 0, semicolon, i less than grades dot length, semicolon, i plus, plus, right parentheses. Line 27, indented twice: left brace. Line 28, indented thrice: if, left parentheses, j less than grades, left square bracket, i, right square bracket, dot, length, right parentheses. Line 29, indented 4 times: System dot out dot print, left parentheses, grades, left square bracket, i, right square bracket, left square bracket, j, right square bracket, right parentheses, semicolon. Line 30, indented thrice: System dot out dot print, left parentheses, double quotes, backslash, t, double quotes, right parentheses, semicolon. Line 31, indented thrice: right brace. Line 32, indented twice: right brace. Line 33, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 34, indented once: right brace. Line 35: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, final, int, double, String, void, for, static. Line 1: Forward slash. Asterisk. A two hyphen dimensional bar chart application. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward salsh. Line 4: blank. Line 5: import java f x dot application dot Application, semicolon. Line 6: import java f x dot stage dot stage, semicolon. Line 7: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 8: import java f x dot scene dot paint dot Color, semicolon. Line 9: blank. Line 10: public class Family Bar Chart Application extends Application. Line 11: left brace. Line 12, indented once: final int LEFT underscore MARGIN equals 80, semicolon. 2 forward slashes, starting x coordinate. Line 13, indented once: final int BASE underscore Y underscore BAR, equals, 100, semicolon. 2 forward slashes, bottom of the bars. Line 14, indented once: final int BASE underscore Y underscore VALUE equals 125, semicolon. 2 forward slashes, bottom of the values. Line 15, indented once: final int BAR underscore WIDTH equals 30, semicolon. 2 forward slashes, width of each bar. Line 16, indented once: final int SPACE underscore BETWEEN underscore BARS equals 10, semicolon. 2 forward slashes, pixels between bars. Line 17, indented once: final int ROW underscore HEIGHT equals 110, semicolon. 2 forward salshes, pixels between rows. Line 18: blank. Line 19, indented once: double, left square bracket, right square bracket, left square bracket, right square bracket, family Cell Bills, equals, left brace, left brace, 45.24, comma, 54.67, comma, 32.55, comma, 25.61, right brace, comma. Line 20, indented multiple times: left brace, 65.29, comma, 49.75, comma, 32.08, comma, 26.11, right brace, comma. Line 21, indented multiple times: left brace, 75.24, comma, 54.53, comma, 34.55, comma, 28.16, right brace, right brace, semicolon. Line 22: blank. Line 23, indented once: String, left square bracket, right square bracket, months, equals, left brace, double quotes, July, double quotes, comma, double quotes, August, double quotes, comma, double quotes, September, double quotes, right brace, semicolon. Line 24, indented once: Color, left square bracket, right square bracket, colors, equals, left brace, Color dot RED, comma, Color dot BLUE, comma, Color dot MAGENTA, right brace, semicolon. Line 25: blank. Line 26, indented once: at symbol, Override. Line 27, indented once: public void start, left parentheses, Stage stage, right parentheses.

Line 28, indented once: left brace. Line 29, indented twice: 2 forward slashes, set up window title and size. Line 30, indented twice: Graphics Context g c, equals, J I Graphics Utility dot set Up Graphics, left parentheses. Line 31, indented multiple times: stage, comma, double quotes, Family Bar Chart, double quotes, comma, 275, comma, 375, right parentheses, semicolon. Line 32: blank. Line 33, indented twice: int x Start, equals, LEFT underscore MARGIN, semicolon. 2 forward slashes, x value for 1 s t column, left parentheses, bars, right parentheses. Line 34, indented twice: int y Start, equals, BASE underscore Y underscore VALUE, semicolon. 2 forward slashes, y value for 1 s t row, left parentheses, data, right parentheses. Line 35, indented twice: int y Start Bar, equals, BASE underscore Y underscore BAR, semicolon. 2 forward slashes, y value for 1 s t row, left parentheses, bars, right parentheses. Line 36: blank. Line 37, indented twice: for, left parentheses, int i equals 0, semicolon, i less than family Cell Bills dot length, semicolon, i plus, plus, right parentheses. Line 38, indented twice: left brace. Line 39, indented thrice: g c dot set Fill, left parentheses, colors, left square bracket, i, right square bracket, right parentheses, semicolon. 2 forward slashes, set color for current row. Line 40, indented thrice: g c dot fill Text, left parentheses, months, left square bracket, i, right square bracket, comma, x Start, minus, LEFT underscore MARGIN, plus, 10, comma. Line 41, indented multiple times: y Start minus decimal point, 3, asterisk, ROW underscore HEIGHT, right parentheses, semicolon. Line 42: blank. Line 43, indented thrice: for, left parentheses, int j equals 0, semicolon, j less than family Cell Bills, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 44, indented thrice: left brace. Line 45, indented 4 times: g c dot fill Rect, left parentheses, x Start, comma, y Start Bar, minus, family Cell Bills, left square bracket, i, right square bracket, left square bracket, j, right square bracket, comma. Line 46, indented multiple times: BAR underscore WIDTH, comma, family Cell Bills, left square bracket, i, right square bracket, left square bracket, j, right square bracket, right parentheses, semicolon. Line 47: blank. Line 48, indented 4 times: g c dot fill Text, left parentheses, Double dot to String, left parentheses, family Cell Bills, left square bracket, i, right square bracket, left square bracket, j, right square bracket, right parentheses, comma. Line 49, indented multiple times: x Start, comma, y Start, right parentheses, semicolon. Line 50: blank. Line 51, indented 4 times: 2 forward slashes, move to starting x value for next bar. Line 52, indented 4 times: x Start, plus, equals, BAR underscore WIDTH, plus, SPACE underscore BETWEEN underscore BARS, semicolon. Line 53, indented thrice: right brace. Line 54: blank. Line 55, indented thrice: 2 forward slashes, new row, colon, increase y Start and y Start Bar. Line 56, indented thrice: y Start, plus, equals, ROW underscore HEIGHT, semicolon. Line 57, indented thrice: y Start Bar, plus, equals, ROW underscore HEIGHT, semicolon. Line 58, indented thrice: x Start, equals, LEFT underscore MARGIN, semicolon. Line 59, indented twice: right brace. Line 60, indented once: right brace. Line 61: blank. Line 62, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 63, indented once: left brace. Line 64, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 65, indented once: right brace. Line 66: right brace.

Back to Example

The syntax is as follows: return Type method Name, left parentheses, array Type, left square bracket, right square bracket, left square bracket, right square bracket, array Parameter Name, right parentheses.

Back to Code

The syntax is as follows: return Array Type, left square bracket, right square bracket, left square bracket, right square bracket, method Name, left parentheses, parameter List, right parentheses.

Back to Code

return Array Type, left square bracket, right square bracket, left square bracket, right square bracket, method Name, left parentheses, array Type, left square bracket, right square bracket, left square bracket, right square bracket, array Parameter Name, right parentheses.

Back to Code

Program code. The code contains the following keywords: public, class, int, this, new, for, return. Line 1: Forward slash. 2 asterisks. Two hyphen Dimensional Arrays as Method Parameters. Line 2: Asterisk. Tab space. and Return Values, colon, the Tally class. Line 3: Asterisk. Tab space. Anderson, comma, Franceschi. Line 4: Asterisk. Forward slash. Line 5: blank. Line 6: public class Tally. Line 7: left brace. Line 8, indented once: int, left square bracket, right square bracket, left square bracket, right square bracket, vote Data, semicolon. Line 9: blank. Line 10, indented once: Forward slash. 2 asterisks, constructor. Line 11, indented once: Asterisk. at symbol, param. Tab space. vote Data a 2 hyphen D array of vote counts. Line 12, indented once: Asterisk. Forward slash. Line 13, indented once: public Tally, left parentheses, int, left square bracket, right square bracket, left square bracket, right square bracket, vote Data, right parentheses. Line 14, indented once: left brace. Line 15, indented twice: this dot vote Data, equals, new int, left square bracket, vote Data dot length, right square bracket, left square bracket, right square bracket, semicolon. Line 16, indented twice: for, left parentheses, int i equals 0, semicolon, i less than vote Data dot length, semicolon, i plus, plus, right parentheses. Line 17, indented thrice: this dot vote Data, left square bracket, i, right square bracket, equals, new int, left square bracket, vote Data, left square bracket, i, right square bracket, dot, length, right square bracket, semicolon. Line 18: blank. Line 19, indented twice: for, left parentheses, int i equals 0, semicolon, i less than vote Data dot length, semicolon, i plus, plus, right parentheses. Line 20, indented twice: left brace. Line 21, indented thrice: for, left parentheses, int j equals 0, semicolon, j less than vote Data, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 22, indented thrice: left brace. Line 23, indented 4 times: this dot vote Data, left square bracket, i, right square bracket, left square bracket, j, right square bracket, equals, vote Data, left square bracket, i, right square bracket, left square bracket, j, right square bracket, semicolon. Line 24, indented thrice: right brace. Line 25, indented twice: right brace. Line 26, indented once: right brace. Line 27: blank. Line 28, indented once: Forward slash. 2 asterisks. array Tally method. Line 29, indented once: Asterisk. at symbol, return a two hyphen dimensional array of votes. Line 30, indented once: Asterisk. Forward slash. Line 31, indented once: public int, left square bracket, right square bracket, left square bracket, right square bracket, array Tally, left parentheses, right parentheses. Line 32, indented once: left brace. Line 33, indented twice: 2 forward slashes, create array of tallies, comma, all elements are 0. Line 34, indented twice: int, left square bracket, right square bracket, left square bracket, right square bracket, return Tally, equals, new int, left square bracket, 2, right square bracket, left square bracket, right square bracket, semicolon. Line 35, indented twice: return Tally, left square bracket, 0, right square bracket, equals, new int, left square bracket, vote Data, left square bracket, 0, right square bracket, dot, length, right square bracket, semicolon. Line 36, indented twice: return Tally, left square bracket, 1, right square bracket, equals, new int, left square bracket, vote Data dot length, right square bracket, semicolon. Line 37: blank. Line 38, indented twice: for, left parentheses, int i equals 0, semicolon, i less than vote Data dot length, semicolon, i plus, plus, right parentheses. Line 39, indented twice: left brace. Line 40, indented thrice: for, left parentheses, int j equals 0, semicolon, j less than vote Data, left square bracket, i, right square bracket, dot length, semicolon, j plus, plus, right parentheses. Line 41, indented thrice: left brace. Line 42, indented 4 times: return Tally, left square bracket, 0, right square bracket, left square bracket, j, right square bracket, plus, equals, vote Data, left square bracket, i, right square bracket, left square bracket, j, right square bracket, semicolon. 2 forward slashes, add to column sum. Line 43, indented 4 times: return Tally, left square bracket, 1, right square bracket, left square bracket, i, right square bracket, plus, equals, vote Data, left square bracket, i, right square bracket, left square bracket, j, right square bracket, semicolon. 2 forward slashes, add to row sum. Line 44, indented thrice: right brace. Line 45, indented twice: right brace. Line 46, indented twice: return return Tally, semicolon. Line 47, indented once: right brace. Line 48: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, int, for. Line 1: Forward slash. 2 asterisks. Tally votes, colon, the Vote Tally class. Line 2: Asterisk. Tab space. Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Vote Tally. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: 2 forward salshes, votes are for 4 candidates in 6 districts. Line 10, indented twice: int, left square bracket, right square bracket, left square bracket, right square bracket, votes, equals, left brace, left brace, 150, comma, 253, comma, 125, comma, 345, right brace, comma. Line 11, indented multiple times: left brace, 250, comma, 750, comma, 234, comma, 721, right brace, comma. Line 12, indented multiple times: left brace, 243, comma, 600, comma, 212, comma, 101, right brace, comma. Line 13, indented multiple times: left brace, 234, comma, 243, comma, 143, comma, 276, right brace, comma. Line 14, indented multiple times: left brace, 555, comma, 343, comma, 297, comma, 990, right brace, comma. Line 15, indented multiple times: left brace, 111, comma, 426, comma, 834, comma, 101, right brace, right brace, semicolon. Line 16, indented twice: 2 forward slashes, candidate names. Line 17, indented twice: String, left square bracket, right square bracket, candidates, equals, left brace, double quotes, Smith, double quotes, comma, double quotes, Jones, double quotes, comma, double quotes, Berry, double quotes, comma, double quotes, Chase, double quotes, right brace, semicolon. Line 19: blank. Line 20, indented twice: 2 forward slashes, instantiate a Tally object reference. Line 21, indented twice: Tally tally, equals, new Tally, left parentheses, votes, right parentheses, semicolon. Line 22: blank. Line 23, indented twice: 2 forward slashes, call array Tally method to count the votes. Line 24, indented twice: int, left square bracket, right square bracket, left square bracket, right square bracket, vote counts, equals, tally dot array Tally, left parentheses, right parentheses, semicolon. Line 25: blank. Line 26, indented twice: 2 forward slashes, print totals for candidates. Line 27, indented twice: System dot out dot print l n, left parentheses, double quotes, Total votes per candidate, double quotes, right parentheses, semicolon. Line 28, indented twice: for, left parentheses, int i equals 0, semicolon, i less than candidates dot length, semicolon, i plus, plus, right parentheses. Line 29, indented thrice: System dot out dot print, left parentheses, candidates, left square bracket, i, right square bracket, plus, double quotes, backslash, t, double quotes, right parentheses, semicolon. Line 30, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 31, indented twice: for, left parentheses, int j equals 0, semicolon, j less than vote Counts, left square bracket, 0, right square bracket, dot length, semicolon, j plus, plus, right parentheses. Line 32, indented thrice: System dot out dot print, left parentheses, vote Counts, left square bracket, 0, right square bracket, left square bracket, j, right square bracket, plus, double quotes, backslash, t, double quotes, right parentheses, semicolon. Line 33, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 34: blank. Line 35, indented twice: 2 forward slashes, print totals for districts. Line 36, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, Total votes per district, double quotes, right parentheses, semicolon. Line 37, indented twice: for, left parentheses, int i equals 0, semicolon, i less than vote Counts, left square bracket, 1, right square bracket, dot length, semicolon, i plus, plus, right parentheses. Line 38, indented thrice: System dot out dot print, left parentheses, left parentheses, i plus 1, right parentheses, plus, double quotes, backslash, t, backslash, t, double quotes, right parentheses, semicolon. Line 39, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 40, indented twice: for, left parentheses, int i equals 0, semicolon, i less than vote Counts, left square bracket, 1, right square bracket, dot length, semicolon, i plus, plus, right parentheses. Line 41, indented thrice: System dot out dot print, left parentheses, vote Counts, left square bracket, 1, right square bracket, left square bracket, i, right square bracket, plus, double quotes, backslash, t, double quotes, right parentheses, semicolon. Line 42, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 43, indented once: right brace. Line 44: right brace.

Back to Example

Line 1: 2 Forward slashes. 5 asterisks, 1. This method has been coded as an example. Line 2: Forward slash, comma, 2 asterisks. Fills the array with random numbers between 50 and 80. Line 3: Asterisk. The instance variable named int Array is the integer array to be. Line 4: Asterisk. filled with values. Line 5: Asterisk, comma, Forward slash. Line 6: public void fill Values Left parentheses, comma, right parentheses. Line 7: Left brace. Line 8: Random rand equals new Random Left parentheses, comma, right parentheses, comma, semicolon. Line 9: for, left parentheses, int row equals 0, comma, semicolon. row left angled bracket, int Array dot length, comma, semicolon. row, plus, plus, right parentheses. Line 10: Left brace. Line 11: System, dot, out, dot, print, left parentheses, row, plus, double quotes, backslash, t, double quotes, right parentheses, comma, semicolon. Line 12: for, left parentheses, int column equals 0, comma, semicolon. j less than int Array, left square bracket, row, right square bracket, dot, length, comma, semicolon. column, plus, plus, right parentheses. Line 13: Left brace. Line 14: int Array, left square bracket, row, right square bracket, left square bracket, column, right square bracket, equals rand dot next Int, left parentheses, 31, right parentheses, plus 50, comma, semicolon. Line 15: animate, left parentheses, row, comma, column, comma, negative 1, right parentheses, comma, semicolon. 2 Forward slashes. needed for visual feedback. Line 16: Right brace. Line 17: System dot out dot print l n, Left parentheses, comma, right parentheses, comma, semicolon. Line 18: Right brace. Line 19: Right brace. 2 Forward slashes. end of fill Values method. Line 20: 2 Forward slashes. 5 asterisks, 2. Student writes this method. Line 21: Forward slash, comma, 2 asterisks. Prints array to the console, comma, elements are separated by a space. Line 22: Asterisk. The instance variable named int Array is the integer array to be. Line 23: Asterisk. printed. Line 24: Asterisk, comma, Forward slash. Line 25: public void print Array Left parentheses, comma, right parentheses. Line 26: Left brace. Line 27: 2 Forward slashes. Note, colon, To animate the algorithm, comma, put this method call as the. Line 28: 2 Forward slashes. last statement in your inner for loop. Line 29: 2 Forward slashes. animate, left parentheses, row, comma, column, comma, minus 1, right parentheses, comma, semicolon. Line 30: 2 Forward slashes. where row is the index of the array's current row. Line 31: 2 Forward slashes. and column is the index of the array's current column. Line 32: 2 Forward slashes. Write your code here, colon. Line 33: Right brace. 2 Forward slashes. end of print Array method. Line 34: 2 Forward slashes. 5 asterisks, 3. Student writes this method. Line 35: Forward slash, comma, 2 asterisks. Sets all the elements in the specified row to the specified value. Line 36: Asterisk. The instance variable named int Array is the integer array.

Line 37: Asterisk. at symbol param value the value to assign to the element of the row. Line 38: Asterisk. at symbol param row the row in which to set the elements to value. Line 39: Asterisk, comma, Forward slash. Line 40: public void set Values, left parentheses, int value, comma, int row, right parentheses. Line 41: Left brace. Line 42: 2 Forward slashes. Note: To animate the algorithm, comma, put this method call as the. Line 43: 2 Forward slashes. last statement in your for loop. Line 44: 2 Forward slashes. animate, left parentheses, row, comma, column, comma, minus 1, right parentheses, comma, semicolon. Line 45: 2 Forward slashes. where row is the index of the array's current row. Line 46: 2 Forward slashes. where column is the index of the array's current column. Line 47: 2 Forward slashes. Write your code here, colon. Line 48: Right brace. 2 Forward slashes. end of set Values method. Line 49: 2 Forward slashes. 5 asterisks, 4. Student writes this method. Line 50: Forward slash, comma, 2 asterisks. Finds minimum value in the specified column. Line 51: Asterisk. The instance variable named int Array is the integer array. Line 52: Asterisk. at symbol param column the column to search. Line 53: Asterisk. at symbol return the minimum value found in the column. Line 54: Asterisk, comma, Forward slash. Line 55: public int find Minimum, left parentheses, int column, right parentheses. Line 56: Left brace. Line 57: 2 Forward slashes. Note: To animate the algorithm, comma, put this method call as the. Line 58: 2 Forward slashes. last statement in your for loop. Line 59: 2 Forward slashes.

animate, left parentheses, row, comma, column, comma, minimum, right parentheses, comma, semicolon. Line 60: 2 Forward slashes. where row is the index of the array's current row. Line 61: 2 Forward slashes. column is the index of the array's current column. Line 62: 2 Forward slashes. minimum is the variable storing the current minimum. Line 63: 2 Forward slashes. Write your code here, colon. Line 64: return 0, comma, semicolon. 2 Forward slashes. replace this line with your return statement. Line 65: Right brace. 2 Forward slashes. end of find Minimum n method. Line 66: 2 Forward slashes. 5 asterisks, 5. Student writes this method. Line 67: Forward slash, comma, 2 asterisks. Finds the number of times value is found in the array. Line 68: Asterisk. The instance variable named int Array is the integer array. Line 69: Asterisk. at symbol param value the value to count. Line 70: Asterisk. at symbol return the number of times value was found. Line 71: Asterisk, comma, Forward slash. Line 72: public int count Found, left parentheses, int value , right parentheses. Line 73: Left brace. Line 74: 2 Forward slashes. Note: To animate the algorithm, comma, put this method call as the. Line 75: 2 Forward slashes. last statement in your inner for loop. Line 76: 2 Forward slashes. animate, left parentheses, row, comma, column, comma, num, right parentheses, comma, semicolon. Line 77: 2 Forward slashes. where row is the index of the array's current row. Line 78: 2 Forward slashes. column is the index of the array's current column. Line 79: 2 Forward slashes. num is the local variable storing the current frequency. Line 80: 2 Forward slashes. count. Line 81: 2 Forward slashes. Write your code here, colon. Line 82: return 0, comma, semicolon. 2 Forward slashes. replace this line with your return statement. Line 83: Right brace. Line 84: 2 Forward slashes. end of count Found method.

Back to Example

animate, left parentheses, row, comma, column, right parentheses, semicolon.

Back to Code

The table has 2 columns labeled Dimension and Array Element. The row entries are as follows. Row 1. Dimension: first. Array Element: array Name, left square bracket, i subscript 1, right square bracket, is an, left parentheses, n minus 1, right parentheses, hyphen, dimensional array. Row 2. Dimension: second. Array Element: array Name, left square bracket, i subscript 1, right square bracket, left square bracket, i subscript 2, right square bracket, is an, left parentheses, n minus 2, right parentheses, hyphen, dimensional array. Row 3. Dimension: k superscirpt t h. Array Element: array Name, left square bracket, i subscript 1, right square bracket, left square bracket, i subscript 2, right square bracket, left square bracket, i subscript 3, right square bracket, left square bracket, dot, dot, dot, right square bracket, left square bracket, i subscript k, right square bracket, is an, left parentheses, n minus k, right parentheses, multi, hyphen, dimensional array. Row 4. Dimension: left square bracket, n minus 2, right square bracket, superscript t h. Array Element: array Name, left square bracket, i subscript 1, right square bracket, left square bracket, i subscript 2, right square bracket, left square bracket, i subscript 3, right square bracket, left square bracket, dot, dot, dot, right square bracket, left square bracket, i subscript n minus 2, right square bracket, is a two, hyphen, dimensional array. Row 5. Dimension: left square bracket, n minus 1, right square bracket, superscript t h. Array Element: array Name, left square bracket, i subscript 1, right square bracket, left square bracket, i subscript 2, right square bracket, left square bracket, i subscirpt 3, right square bracket, left square bracket, dot, dot, dot, right square bracket, left square bracket, i subscript n minus 1, right square bracket, is a single hyphen dimensional array. Row 6. Dimension: n superscript t h. Array Element: array Name, left square bracket, i subscript 1, right square bracket, left square bracket, i subscript 2, right square bracket, left square bracket, i subscript 3, right square bracket, left square bracket, dot, dot, dot, right square bracket, left square bracket, i subscript n minus 1, right square bracket, left square bracket, i subscript n, right square bracket, is an array element.

Back to Table

Line 1: double, left square bracket, right square bracket, left square bracket, right square bracket, left square bracket, right square bracket, sales, semicolon. 2 forward slashes, declare a four hyphen dimensional array. Line 2: Sales equals new double, left square bracket, 10, right square bracket, left square bracket, 52, right square bracket, left square bracket, 7, right square bracket, semicolon. 2 forward slashes, instantiate the array. Line 3: Sales, left square bracket, 0, right square bracket, left square bracket, 0, right square bracket, left square bracket, 0, right square bracket, equals, 638.50, semicolon, 2 forward slashes, access the first element. Line 4: sales, left square bracket, 4, right square bracket, left square bracket, 22, right square bracket, left square bracket, 3, right square bracket, equals, 928.30, semicolon. 2 forward slashes, access another element. Line 5: sales, left square bracket, 9, right square bracket, left square bracket, 51, right square bracket, left square bracket, 6, right square bracket, equals, 1234.90, semicolon. 2 forward slashes, access the last element.

Back to Code

Program code. The code contains the following keywords: for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i less than array Name dot length, semicolon, i plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int j equals 0, semicolon, j less than array Name, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 4, indented once: left brace. Line

5, indented twice: for, left parentheses, int k equals 0, semicolon, k less than array Name, left square bracket, i, right square bracket, left square bracket, j, right square bracket, dot, length, semicolon, k plus, plus, right parentheses. Line 6, indented twice: left brace. Line 7, indented thrice: 2 forward slashes, access and process the element array Name, left square bracket, i, right square bracket, left square bracket, j, right square bracket, left square bracket, k, right square bracket. Line 8, indented twice: right brace. Line 9, indented once: right brace. Line 10: right brace.

Back to Code

Program code. The code contains the following keywords: for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i less than sales dot length, semicolon, i plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int j equals 0, semicolon, j less than sales, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: for, left parentheses, int k equals 0, semicolon, k less than sales, left square bracket, i, right square bracket, left square bracket, j, right square bracket, dot, length, semicolon, k plus, plus, right parentheses. Line 6, indented twice: left brace. Line 7, indented twice: 2 forward slashes, access the element at sales, left square bracket, i, right square bracket, left square bracket, j, right square bracket, left square bracket, k, right square bracket. Line 8, indented twice: System dot out dot print l n, left parentheses, sales, left square bracket, i, right square bracket, left square bracket, j, right square bracket, left square bracket, k, right square bracket, plus, double quotes, back slash t, double quotes, right parentheses, semicolon. Line 9, indented twice: right brace. Line 10, indented twice: 2 forward slashes, skip a line when second dimension index changes. Line 11, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 12, indented once: right brace. Line 13, indented once: 2 forward slashes, skip a line when first dimension index changes. Line 14, indented once: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 15: right brace.

Back to Code

Line 1: double, left square bracket, right square bracket, left square bracket, right square bracket, left square bracket, right square bracket, left square bracket, right square bracket, state Sales, semicolon. 2 forward slashes, declare a four hyphen dimensional. Line 2, indented multiple times: 2 forward slashes, array. Line 3: state Sales, equals, new double, left square bracket, 50, right square bracket, left square bracket, 10, right square bracket, left square bracket, 52, right square bracket, left square bracket, 7, right square bracket, semicolon. 2 forward slashes, instantiate the array. Line 4: state Sales, left square bracket, 0, right square bracket, left square bracket, 0, right square bracket, left square bracket, 0, right square bracket, left square bracket, 0, right square bracket, equals, 58.50, semicolon. 2 forward slashes, access the first element. Line 5: sales, left square bracket, 34, right square bracket, left square bracket, 4, right square bracket, left square bracket, 22, right square bracket, left square bracket, 3, right square bracket, 98.30, semicolon. 2 forward slashes, access another element. Line 6: sales, left square bracket, 49, right square bracket, left square bracket, 9, right square bracket, left square bracket, 51, right square bracket, left square bracket, 6, right square bracket, equals, 137.70, semicolon. 2 forward slashes, access the last element.

Back to Code

Program code. The code contains the following keywords: for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i less than array Name dot length, semicolon, i plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int j equals 0, semicolon, j less than array Name, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: for, left parentheses, int k equals 0, semicolon, k less than array Name, left square bracket, i, right square bracket, left square bracket, j, right square bracket, dot, length, semicolon, k plus, plus, right parentheses. Line 6, indented twice: left brace. Line 7, indented thrice: for, left parentheses, int l equals 0, semicolon, l less than array Name, left square bracket, i, right square bracket, left square bracket, j, right square bracket, left square bracket, k, right square bracket, dot, length, semicolon, l plus, plus, right parentheses. Line 8, indented thrice: left brace. Line 9, indented four times: 2 forward slashes, process element array Name, left square bracket, i, right square bracket, left square bracket, j, right square bracket, left square bracket, k, right square bracket, left square bracket, l, right square bracket. Line 10, indented thrice: right brace. Line 11, indented twice: right brace. Line 12, indented once: right brace. Line 13: right brace.

Back to Code

import java dot util dot Array List, semicolon.

Back to Code

The syntax is as follows: Array List, left angled bracket, Class Name, right angled bracket, array List Name, semicolon.

Back to Code

Line 1: Array List, left angled bracket, String, right angled bracket, list Of Strings, semicolon. Line 2: Array List, left angled bracket, Auto, right angled bracket, list Of Cars, semicolon.

Back to Code

Array List, left angled bracket, int, right angled bracket, list

Of Ints, semicolon.

Back to Code

Line 1: Test dot java, colon, 7, colon, error, colon, unexpected type. Line 2, indented once: Array List, left angled bracket, int, right angled bracket, list Of Ints, semicolon. Line 3: a caret sign under lowercase i following the left angled bracket in Line 2. Line 4: required, colon, reference. Line 5: found, colon, int. Line 6: 1 error.

Back to Code

The title of the table reads: Array List Constructors Summary. The constructor name and argument list are as follows. Row 1. Array List, left angled bracket, Class Name, right angled bracket, left parentheses, right parentheses, constructs an Array List object of Class Name type with an initial capacity of 10. Row 2. Array List, left angled bracket, Class Name, right angled bracket, left parentheses, int initial Capacity, right parentheses, constructs an Array List object of Class Name type with the specified initial capacity.

Back to Table

The syntax is as follows. array List Name, equals, new, Array List, left angled bracket, Class Name, right angled bracket, left parentheses, right parentheses, semicolon. The caption below the syntax is as follows: where Class

Name is the class type of the objects that will be stored in the Array List and array List Name has been declared previously as an Array List reference for that class.

Back to Code

Line 1: list Of Strings, equals, new Array List, left angled bracket, String, right angled bracket, left parentheses, right parentheses, semicolon. Line 2: list Of Cars, equals, new Array List, left angled bracket, Auto, right angled bracket, left parentheses, right parentheses, semicolon.

Back to Code

list Of Cars, equals, new, Array List, left parentheses, right parentheses, semicolon.

Back to Code

Line 1: Test dot java, colon, 11, colon, warning, colon, left square bracket, raw types, right square bracket, found raw type, colon, Array List. Line 2, indented once: list Of Cars, equals, new Array List, left parentheses, right parentheses, semicolon. Line 3: a caret sign under uppercase A in Line 2. Line 4, indented once: missing type arguments for generic class Array List, left angled bracket, uppercase e, right angled bracket. Line 5, indented once: where uppercase e is a type hyphen variable, colon. Line 6, indented twice: uppercase e extends object declared in class Array List. Line 7: blank. Line 8: Test dot java, colon, 11, colon, warning, colon, left square bracket, unchecked, right square bracket, unchecked conversion. Line 12, indented once: list Of Cars, equals, new Array List, left parentheses, right parentheses, semicolon. Line 13: a caret sign under lowercase n of new in Line 12. Line 14, indented once: required, colon, Array List, left angled bracket, Auto, right angled bracket. Line 15, indented once: found, colon, Array List. Line 16: 2 warnings.

Back to Code

Line 1: Array List, left angled bracket, String, right angled bracket, list Of Strings, 1, semicolon. Line 2: list Of Strings 1, equals, new, Array List, left angled bracket, String, right angled bracket, left parentheses, 5, right parentheses, semicolon.

Back to Code

The syntax is as follows: Array List, left angled bracket, Class Name, right angled bracket, array List Name, equals, new, Array List, left angled bracket, Class Name, right angled bracket, left parentheses, right parentheses, semicolon.

Back to Code

Line 1: Array List, left angled bracket, Integer, right angled bracket, list Of Ints, equals, new, Array List, left angled bracket, Integer, right angled bracket, left parentheses, right parentheses, semicolon. Line 2: Array List, left angled bracket, String, right angled bracket, list Of Strings 2, equals, new Array List, left angled bracket, String, right angled bracket, left parentheses, right parentheses, semicolon.

Back to Code

The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: boolean. Method name and argument list: add, left parentheses, uppercase e, underlined, element, right parentheses, appends the specified element to the end of the list. Row 2. Return value: uppercase e, underlined. Method name and argument list: remove, left parentheses, int index, right parentheses, removes and returns the element at the specified index position in the list. Row 3. Return value: void. Method name and argument list: clear, left parentheses, right parentheses, removes all the elements from this list. Row 4. Return value: uppercase e underlined. Method name and argument list: get, left parentheses, int index, right parentheses, returns the element at the specified index position in the list; the element is not removed from the list. Row 5. Return value: uppercase e underlined. Method name and argument list: set, left parentheses, int index, comma, uppercase e underlined element, right parentheses, replaces the element at the specified index position in this list with the specified element. Row 6. Return value: int. Method name and argument list: size, left parentheses, right parentheses, returns the number of elements in this list. Row 7. Return value: void. Method name and argument list: trim To Size, left parentheses, right parentheses, sets the capacity to the list's current size.

Back to Table

Program code. The code contains the following keywords: for, int. Line 1: Class Name current Object, semicolon. Line 2: for, left parentheses, int i equals 0, semicolon, i less than array List Name dot size, left parentheses, right parentheses, semicolon, i plus, plus, right parentheses. Line 3: left brace. Line 4, indented once: current Object equals array List Name dot get, left parentheses, i, right parentheses, semicolon. Line 5, indented once: 2 forward slashes, process current Object. Line 6: right brace.

Back to Code

Program code. The code contains the following keywords: String, for, int. Line 1: String current String, semicolon. Line 2: for, left parentheses, int i equals 0, semicolon, i less than list Of Strings dot size, left parentheses, right parentheses, semicolon, i plus, plus, right parentheses. Line 3: left brace. Line 4, indented once: current String, equals, list Of Strings dot get, left parentheses, i, right parentheses, semicolon. Line 5, indented once: 2 forward slashes, process current String. Line 6: right brace.

Back to Code

Program code. The code contains the keyword for. Line 1: for, left parentheses, Class Name current Object, colon, array List Name, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes. process current Object. Line 4: right brace.

Back to Code

Program code. The code contains the keyword for, String. Line 1: for, left parentheses, String current String, colon, names, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes. process current String. Line 4: right brace.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, String, for, int. Line 1: Forward slash. Asterisk. A Simple Array List of Integers. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Array List, semicolon. Line 6: blank. Line 7: public class Array List Of Integers. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Array List, left angled bracket, Integer, right angled bracket, list, equals, new Array List, left angled bracket, Integer, right angled bracket, left parentheses, right parentheses, semicolon.

Line 12, indented twice: list dot add, left parentheses, 34, right parentheses, semicolon. 2 forward slashes, autoboxing. Line 13, indented twice: list dot add, left parentheses, 89, right parentheses, semicolon. 2 forward slashes, autoboxing. Line 14, indented twice: list dot add, left parentheses, 65, right parentheses, semicolon. 2 forward slashes, autoboxing. Line 15: blank. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, Using the traditional for loop, colon, double quotes, right parentheses, semicolon. Line 17, indented twice: for, left parentheses, int i equals 0, semicolon, i less than list dot size, left parentheses, right parentheses, semicolon, i plus, plus, right parentheses. Line 18, indented thrice: System dot out dot print, left parentheses, list dot get, left parentheses, i, right parentheses, plus, double quotes, backslash, t, double quotes, right parentheses, semicolon. Line 19, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 20: blank. Line 21, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, Using the enhanced for loop, colon, double quotes, right parentheses, semicolon. Line 22, indented twice: for, left parentheses, Integer current Integer, colon, list, right parentheses. Line 23, indented thrice: System dot out dot print, left parentheses, current Integer, plus, double quotes, backslash, t, double quotes, right parentheses, semicolon. Line 24, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 25: blank. Line 26, inented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, Using unboxing and enhanced for loop, colon, double quotes, right parentheses, semicolon. Line 27, indented twice: for, left parentheses, int current Int, colon, list, right parentheses, 2 forward slashes, unboxing. Line 28, indented thrice: System dot out dot print, left parentheses, current Int, plus, double quotes, backslash, t, double quotes, right parentheses, semicolon. Line 29, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 30: blank. Line 31, indented twice: list dot set, left parentheses, 1, comma, 100, right parentheses, semicolon. Line 32, indented twice: System dot out dot print l n, let parentheses, double quotes, backslash, n, After calling set, left parentheses, 1, comma, 100, right parentheses, colon, double quotes, right parentheses, semicolon. Line 33, indented twice: for, left parentheses, int current Int, colon, list, right parentheses. 2 forward slashes, unboxing. Line 34, indented thrice: System dot out dot print, left parentheses, current Int, plus, double quotes, backslash, t, double quotes, right parentheses, semicolon. Line 35, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 36: blank. Line 37, indented twice: int removed, equals, list dot remove, left parentheses, 0, right parentheses, semicolon. Line 38, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, At index 0, comma, blank space, double quotes, plus, removed, plus, double quotes, was removed, double quotes, right parentheses, semicolon. Line 39, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, After removing the element at index 0, colon, double quotes, right parentheses, semicolon. Line 40, indented twice: for, left parentheses, int current Int, colon, list, right parentheses, 2 forward slashes, unboxing. Line 41, indented thrice: System dot out dot print, left parentheses, current Int, plus, double quotes, backslash, t, double quotes, right parentheses, semicolon. Line 42, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 43, indented once: right brace. Line 44: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, private, String, double, this, return. Line 1: Forward slash. Asterisk. Book class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: pubilc class Book. Line 6: left brace. Line 7, indented once: private String title, semicolon. Line 8, indented once: private String author, semicolon. Line 9, indented once: private double price, semicolon. Line 10: blank. Line 11, indented once: Forward slash. 2 Asterisks. default constructor. Line 12, indented once: Asterisk. Forward slash. Line 13, indented once: public Book, left parentheses, right parentheses. Line 14, indented once: left brace. Line 15, indented twice: title, equals, double quotes, double quotes, semicolon. Line 16, indented twice: author, equals, double quotes, double quotes, semicolon. Line 17, indented twice: price, equals, 0.0, semicolon. Line 18, indented once: right brace. Line 19: blank. Line 20, indented once: Forward slash. 2 asterisks. overloaded constructor. Line 21, indented once: Asterisk, at symbol, param, titled, the value to assign to title. Line 22, indented once: Asterisk. at symbol, param, author, the value to assign to author. Line 23, indented once: Asterisk, at param, price, the value to assign to price. Line 24, indented once: Asterisk. Forward slash. Line 25, indented once: public Book, left parentheses, String title, comma, String author, comma, double price, right parentheses. Line 26, indented once: left brace. Line 27, indented twice: this dot title, equals, title, semicolon. Line 28, indented twice: this dot author, equals, author, semicolon. Line 29, indented twice: this dot price, equals, price, semicolon. Line 30, indented once: right brace. Line 31: blank. Line 32, indented once: Forward slash. 2 asterisks. get Title method. Line 33, indented once: Asterisk. at symbol, return the title. Line 34, indented once: Asterisk. Forward slash. Line 35, indented once: public String get Title, left parentheses, right parentheses. Line 36, indented once: left brace. Line 37, indented twice: return title, semicolon. Line 38, indented once: right brace. Line 39: blank. Line 40, indented once: Forward slash. 2 asterisks, get Author method. Line 41, indented once: Asterisk, at symbol, return the author. Line 42, indented once: Asterisk. Forward slash. Line 43, indented once: public String get

Author, left parentheses, right parentheses. Line 44, indented once: left brace. Line 45, indented twice: return author, semicolon. Line 46, indented once: right brace. Line 47: blank. Line 48, indented once: Forward slash. 2 asterisks. get Price method. Line 49, indented once: Asterisk. at return the price. Line 50, indented once: Asterisk. Forward slash. Line 51, indented once: public double get Price, left parentheses, right parentheses. Line 52, indented once: left brace. Line 53, indented twice: return price, semicolon. Line 54, indented once: right brace. Line 55: blank. Line 56, indented once: Forward slash. 2 asterisks. to String. Line 57, indented once: Asterisk. at symbol, return, title, comma, author, comma, and price. Line 58, indented once: Asterisk. Forward salsh. Line 59, indented once: at symbol, Override. Line 60, indented once: public String to String, left parentheses, right parentheses. Line 61, indented once: left brace. Line 62, indented twice: return, left parentheses, double quotes, title, colon, double quotes, plus, title, plus, double quotes, backslash, t, double quotes. Line 63, indented multiple times: plus, double quotes, author, colon, double quotes, plus, author, plus, double quotes, backslash, t, double quotes. Line 64, indented multiple times: plus, double quotes, price, colon, double quotes, plus, price, right parentheses, semicolon. Line 65, indented once: right brace. Line 66: right brace.

Back to Example

Program code. The code contains the following keywords:

import, public, class, private, new, String, for, return, if. Line 1: Forward slash. Asterisk. Book Store class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Array List, semicolon. Line 6: blank. Line 7: public class Book Store. Line 8: left brace. Line 9, indented once: private Array List, left angled bracket, Book, right angled bracket, library, semicolon. Line 10: blank. Line 11, indented once: Forward slash. 2 asterisks. default constructor. Line 12, indented once: Asterisk. Tab space. instantiates Array List of Books. Line 13, indented once: Asterisk. Forward slash. Line 14, indented once: public Book Store, left parentheses, right parentheses. Line 15, indented once: left brace. Line 16, indented twice: library, equals, new, Array List, left angled bracket, Book, right angled bracket, left parentheses, right parentheses, semicolon. Line 17, indented twice: library dot add, left parentheses, new Book, left parentheses, double quotes, Intro to Java, double quotes, comma, double quotes, James, double quotes, comma, 56.99, right parentheses, right parentheses, semicolon. Line 18, indented twice: library dot add, left parentheses, new Book, left parentheses, double quotes, Advanced Java, double quotes, comma, double quotes, Green, double quotes, comma, 65.99, right parentheses, right parentheses, semicolon. Line 19, indented twice: library dot add, left parentheses, new Book, left parentheses, double quotes, Java Servlets, double quotes, comma, double quotes, Brown, double quotes, comma, 75.99, right parentheses, right parentheses, semicolon. Line 20, indented twice: library dot add, left parentheses, new Book, left parentheses, double quotes, Intro to H T M L, double quotes, comma, double quotes, James, double quotes, comma, 29.49, right parentheses, right parentheses, semicolon. Line 21, indented twice: library dot add, left parentheses, new Book, left parentheses, double quotes, Intro to Flash, double quotes, comma, double quotes, James, double quotes, comma, 34.99, right parentheses, right parentheses, semicolon. Line 22, indented twice: library dot add, left parentheses, new Book, left parentheses, double quotes, Advanced H T M L, double quotes, comma, double quotes, Green, double quotes, comma, 56.99, right parentheses, right parentheses, semicolon. Line 23, indented twice: library dot trim To Size, left parentheses, right parentheses, semicolon. Line 24, indented twice: right brace. Line 25: blank. Line 26, indented once: Forward slash. 2 asterisks. to String. Line 27, indented once: Asterisk. at symbol, return each book in library, comma, one per line. Line 28, indented once: Asterisk. Forward slash. Line 29, indented once: at symbol, Override. Line 30, indented once: public String to String, left parentheses, right parentheses. Line 31, indented once: left brace. Line 32, indented twice: String result, equals, double quotes, double quotes, semicolon. Line 33, indented twice: for, left parentheses, Book temp Book, colon, library, right parentheses. Line 34, indented twice: left brace. Line 35, indented thrice: result, plus, equals, temp Book dot to String, left parentheses, right parentheses, plus, double quotes, backslash, n, double quotes, semicolon. Line 36, indented twice: right brace. Line 37, indented twice: return result, semicolon. Line 38, indented once: right brace. Line 39: blank. Line 40, indented once: Forward slash. 2 asterisks. Generates list of books containing search String. Line 41, indented once: Asterisk, at symbol, param, search String. Tab space. the keyword to search for. Line 42, indented once: Asterisk. at symbol, return. Tab space. the Array List of books containing the keyword. Line 43, indented once: Asterisk. Forward slash. Line 44, indented once: public Array List, left angled bracket, Book, right angled bracket, search For Title, left parentheses, String search String, right parentheses. Line 45, indented once: left brace. Line 46, indented twice: Array List, left angled bracket, Book, right angled bracket, search Result, equals, new Array List, left angled bracket, Book, right angled bracket, left parentheses, right parentheses, semicolon. Line 47, indented twice: for, left parentheses, Book current Book, colon, library, right parentheses. Line 48, indented twice: left brace. Line 49, indented thrice: if, left parentheses, current Book dot get Title, left parentheses, right parentheses, dot, index Of, left parentheses, search String, right parentheses, exclamation mark, equals, negative 1, right parentheses. Line 50, indented multiple times: search Result dot add, left parentheses, current Book, right parentheses, semicolon. Line 51, indented twice: right brace. Line 52, indented twice: search Result dot trim To Size, left parentheses, right parentheses, semicolon. Line 53, indented twice: return search Result, semicolon. Line 54, indented once: right brace. Line 55: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, new, for. Line 1: Forward slash. Asterisk. Book Search Engine class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot asterisk, semicolon. Line 6: blank. Line 7: public class Book Search Engine. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, string, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Book Store b s, equals, new Book Store, left parentheses, right parentheses, semicolon. Line 12: blank. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, Our book collection is, colon, double quotes, right parentheses, semicolon. Line 14, indented twice: System dot out dot print l n, left parentheses, b s dot to String, left parentheses, right parentheses, right parentheses, semicolon. Line 15, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 16, indented twice: System dot out dot print, left parentheses, double quotes, Enter a keyword, right angled bracket, double quotes, right parentheses, semicolon. Line 17, indented twice: String keyword, equals, scan dot next, left parentheses, right parentheses, semicolon. Line 18: blank. Line 19, indented twice: Array List, left angled bracket, Book, right angled bracket, results, equals, b s dot search For Title, left parentheses, keyword, right parentheses, semicolon. Line 20: blank. Line 21, indented twice: System dot out dot print l n, left parentheses, double quotes, The search results for, double quotes, plus, keyword. Line 22, indented multiple times: plus, double quotes, are, colon, double quotes, right parentheses, semicolon. Line 23, indented twice: for, left parentheses, Book temp Book, colon, results, right parentheses. Line 24, indented thrice: System dot out dot print l n, left parentheses, temp Book dot to String, left parentheses, right parentheses, right parentheses, semicolon. Line 25, indented once: right brace. Line 26: right brace.

Back to Example

The first table is titled, Auto Class Constructor Summary. The row entries are as follows. Row 1. Auto, left parentheses, right parentheses, creates an Auto object with initial default values of, double quotes, unknown, double quotes, 0, and 0.0. Row 2. Auto, left parentheses, String model, comma, int miles Driven, comma, double gallons Of Gas, right parentheses, creates an Auto object with the initial values of model, miles Driven, and gallons Of Gas. The second table is titled, Auto Class Method Summary. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: String. Method name and argument list: get Model, left parentheses, right parentheses, returns the value of model. Row 2. Return value: int. Method name and argument list: get Miles Driven, left parentheses, right parentheses, returns the value of miles Driven. Row 3. Return value: double. Method name and argument list: get Gallons Of Gas, left parentheses, right parentheses, returns the value of gallons Of Gas. Row 4. Return value: Auto. Method name and argument list: set Model, left parentheses, String model, right parentheses, mutator for model. Row 5. Return value: Auto. Method name and argument list: set Miles Driven, left parentheses, int miles Driven, right parentheses, mutator for miles Driven. Row 6. Return value: Auto. Method name and argument list: set Gallons Of Gas, left parentheses, double gallons Of Gas, right parentheses, mutator for gallons Of Gas. Row 7. Return value: double. Method name and argument list: miles Per Gallon, left parentheses, right parentheses, returns the gas mileage for this Auto object. Row 8. Return value: String. Method name and argument list: to String, left parentheses, right parentheses, returns a String representation of this object. Row 9. Return value: boolean. Method name and argument list: equals, left parentheses, Object o b j, right parentheses, compares this Auto object to another object.

Back to Table

Line 1: 2 Forward slashes. 5 asterisks, 1. This method has been coded as an example. Line 2: Forward slash, comma, 2 asterisks. Fills the car List with hard hyphen coded Auto objects. Line 3: Asterisk. The instance variable car List is the Array List. Line 4: Asterisk. to be filled with Auto objects. Line 5: Asterisk, comma, Forward slash. Line 6: public void fill With Cars, Left parentheses, comma, right parentheses. Line 7: Left brace. Line 8: 2 Forward slashes. clear car List before adding cars. Line 9: car List dot clear, Left parentheses, comma, right parentheses, comma, semicolon. Line 10: 2 Forward slashes. Reset the number of Autos to 0. Line 11: 2 Forward slashes. This is needed so that the animation feedback works correctly. Line 12: Auto dot clear Number Autos, Left parentheses, comma, right parentheses, comma, semicolon. Line 13: Auto car 1 equals new Auto, left parentheses, double quotes, B M W double quotes, comma, 0, comma, 0.0, right parentheses, comma, semicolon. Line 14: Auto car 2 equals new Auto, left parentheses, double quotes, Ferrari double quotes, comma, 100, comma, 500.0, right parentheses, , comma, semicolon. Line 15: Auto car 3 equals new Auto, left parentheses, double quotes, Jeep double quotes, , comma, 1000, comma, 90.0, right parentheses, comma, semicolon. Line 16: Auto car 4 equals new Auto, left parentheses, double quotes, Ferrari double quotes, comma, 10, comma, 3.0 , right parentheses, comma, semicolon. Line 17: Auto car 5 equals new Auto, left parentheses, double quotes, BMW double quotes, comma, 4000, comma, 200.0, right parentheses, comma, semicolon. Line 18: Auto car 6 equals new Auto, left parentheses, double quotes, Ferrari double quotes, comma, 1000, comma, 50.0, right parentheses, comma, semicolon. Line 19: car List dot add, left parentheses, car 1, right parentheses, comma, semicolon. Line 20: animate, left parentheses, car 1, right parentheses, comma, semicolon. Line 21: car List dot add, left parentheses, car 2, right parentheses, comma, semicolon. Line 22: animate, left parentheses, car 2, right parentheses, comma, semicolon. Line 23: car List dot add, left parentheses, car 3, right parentheses, comma, semicolon. Line 24: animate, left parentheses, car 3, right parentheses, comma, semicolon. Line 25: car List dot add, left parentheses, car 4, right parentheses, comma, semicolon. Line 26: animate, left parentheses, car 4, right parentheses, comma, semicolon. Line 27: car List dot add, left parentheses, car 5, right parentheses, comma, semicolon. Line 28: animate, left parentheses, car 5, right parentheses, comma, semicolon. Line 29: car List dot add, left parentheses, car 6, right parentheses, comma, semicolon. Line 30: animate, left parentheses, car 6, right parentheses, comma, semicolon. Line 31: Right brace. Line 32: 2 Forward slashes. end of fill With Cars method. Line 33: 2 Forward slashes. 5 asterisks, 2. Student writes this method. Line 34: Forward slash, comma, 2 asterisks. Prints car List to console, comma, elements are separated by a space. Line 35: Asterisk. The instance variable car List is the Array List to be printed. Line 36: Asterisk, comma, Forward slash. Line 37: public void print

Auto List, Left parentheses, comma, right parentheses. Line 38: Left brace. Line 39: 2 Forward slashes. Note: To animate the algorithm, comma, put this method call as the. Line 40: 2 Forward slashes. last statement in your for loop. Line 41: 2 Forward slashes. animate, left parentheses, car, right parentheses, comma, semicolon. Line 42: 2 Forward slashes. where car is the variable name for the current Auto object. Line 43: 2 Forward slashes. as you loop through the Array List object. Line 44: 2 Forward slashes. Write your code here, colon. Line 45: Right brace. Line 46: 2 Forward slashes. end of print Auto List method. Line 47: 2 Forward slashes. 5 asterisks, 3. Student writes this method. Line 48: Forward slash, comma, 2 asterisks. Sets the model of all the elements in car List to parameter. Line 49: value. Line 50: Asterisk. The instance variable car List is the Array List to be modified. Line 51: Asterisk. at symbol param model the model to assign to all Auto objects in car List. Line 52: Asterisk, comma, Forward slash. Line 53: public void set Model Values, left parentheses, String model, right parentheses. Line 54: Left brace. Line 55: 2 Forward slashes. Note: To animate the algorithm, comma, put this method call as the. Line 56: 2 Forward slashes. last statement in your for loop. Line 57: 2 Forward slashes. animate, left parentheses, car, right parentheses, comma, semicolon. Line 58: 2 Forward slashes. where car is the variable name for the current Auto object. Line 59: 2 Forward slashes. as you loop through the Array List object. Line 60: 2 Forward slashes. Write your code here, colon. Line 61: Right brace. Line 62: 2 Forward slashes. end of set Model Values method. Line 63: 2 Forward slashes. 5 asterisks, 4. Student writes this method. Line 64: Forward slash, comma, 2 asterisks. Finds maximum number of miles driven. Line 65: Asterisk. Instance variable car List is the Array List to search. Line 66: Asterisk. at symbol return the maximum miles driven by all the Auto objects. Line 67: Asterisk, comma, Forward slash. Line 68: public int find Maximum Miles Driven, Left parentheses, comma, right parentheses. Line 69: Left brace. Line 70: 2 Forward slashes. Note: To animate the algorithm, comma, put this method call as the. Line 71: 2 Forward slashes. last statement in your for loop. Line 72: 2 Forward slashes. animate, left parentheses, car, comma, maximum, right parentheses, comma, semicolon. Line 73: 2 Forward slashes. where car is the variable name for the current Auto object. Line 74: 2 Forward slashes. and maximum is the int variable storing the current maximum. Line 75: 2 Forward slashes. number of miles for all Auto elements you have already tested. Line 76: 2 Forward slashes. as you loop through the Array List object. Line 77: 2 Forward slashes. Write your code here, colon. Line 78: return 0, comma, semicolon. 2 Forward slashes. replace this statement with your return statement. Line 79: Right brace. Line 80: 2 Forward slashes. end of find Maximum Miles Driven method. Line 81: 2 Forward slashes. 5 asterisks, 5. Student writes this method. Line 82: Forward slash, comma, 2 asterisks. Finds number of times parameter model is found in the car List. Line 83: Asterisk. Instance variable car List is the Array List in which we search. Line 84: Asterisk. at symbol param model the model to count. Line 85: Asterisk. at symbol return the number of times model was found. Line 86: Asterisk, comma, Forward slash. Line 87: public int count Found, left parentheses, String model, right parentheses. Line 88: Left brace. Line 89: 2 Forward slashes. Note, colon, To animate the algorithm, comma, put this method call as the. Line 90: 2 Forward slashes. last statement in your for loop. Line 91: 2 Forward slashes. animate, left parentheses, car, comma, num , right parentheses, comma, semicolon. Line 92: 2 Forward slashes. where car is the variable name for the current Auto object. Line 93: 2 Forward slashes. and num is the int variable storing the current number of. Line 94: 2 Forward slashes. Auto elements whose model is equal to the method's. Line 95: parameter. Line 96: 2 Forward slashes. as you loop through the Array List object. Line 97: 2 Forward slashes. Write your code here, colon. Line 98: return 0, comma, semicolon. 2 Forward slashes. replace this statement with your return statement. Line 99: Right brace. Line 100: 2 Forward slashes. end of count Found method.

Back to Example

Line 1: animate, left parentheses, car, right parentheses, semicolon. 2 forward slashes, or. Line 2: animate, left parentheses, car, comma, maximum, right parentheses, semicolon. 2 forward slashes, or. Line 3: animate, left parentheses, car, comma, num, right parentheses, semicolon.

Back to Code

The syntax is as follows: Hash Map, left angled bracket, Class Name 1, comma, Class Name 2, right angled bracket, hash Map Name, semicolon.

Back to Code

Line 1: Hash Map, left angled bracket, String, comma, String, right angled bracket, countries, semicolon. Line 2: Hash Map, left angled bracket, Integer, comma, Auto, right angled bracket, cars, semicolon.

Back to Code

The table lists the constructor name and argument list. The row entries are as follows. Row 1. Hash Map, left angled bracket, Class Name 1, comma, Class Name 2, right angled bracket, left parentheses, right parentheses. Constructs a Hash Map object whose keys are Class Name 1 objects and values are Class Name 2 objects. Row 2. Hash Map, left angled bracket, Class Name 1, comma, Class Name 2, right angled bracket, left parentheses, int initial Capacity, right parentheses. Constructs a Hash Map object whose keys are Class Name 1 objects and values are Class Name 2 objects. The resulting Hash Map object has teh specified initial capacity and a default load factor of 0.75. A load factor of

0.75 means that if the capacity of the Hash Map is 75 percent full, the capacity will be automatically increased.

Back to Table

The syntax is as follows. hash Map Name, equals, new Hash Map, left angled bracket, Class Name 1, comma, Class Name 2, right angled bracket, left parentheses, argument list, right parentheses, semicolon.

Back to Code

Line 1. countries, equals, new Hash Map, left angled bracket, String, comma, String, right angled bracket, left parentheses, right parentheses, semicolon. Line 2. cars, equals, new Hash Map, left angled bracket, String, comma, Auto, right angled bracket, left parentheses, 10, right parentheses, semicolon.

Back to Code

The title of the table is Useful Methods of the Hash Map Class. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: uppercase v. Method name and argument list: put, left parentheses, uppercase k key, comma, uppercase v value, right parentheses, Adds the, left parentheses, key, comma, value, right parentheses, pair specified to this Hash Map. If this Hash Map previously contained key, the old value is replaced and returned. Returns null if there was no previous mapping for key. uppercase k and uppercase v are the data types for the keys and the values of this Hash Map, respectively. Row 2. Return value: uppercase v. Method name and argument list: get, left parentheses, uppercase k key, right parentheses, Returns the value mapped to key, null if there is no value mapped to key. Row 3. Return value: boolean. Method name and argument list: is Empty, left parentheses, right parentheses, Returns true if this Hash Map contains no mapping, false otherwise. Row 4. Return value: uppercase v. Method name and argument list: remove, left parentheses, uppercase k key, right parentheses, Removes the, left parentheses, key, comma, value, right parentheses, pair if key is found and returns the value mapped to key. Returns null if no mapping exists for key. Row 5. Return value: boolean. Method name and argument list: replace, left parentheses, uppercase k key, comma, uppercase v old Value, comma, uppercase v new Value, right parentheses, changes old Value to new Value if old Value is mapped to key in this Hash Map; if it is, it returns true, false otherwise. Row 6. Return value: int. Method name and argument list: size, left parentheses, right parentheses, Returns the number of, left parentheses, key, comma, value, right parentheses, pairs in this Hash Map. Row 7. Return value: boolean. Method name and argument list: contains, left parentheses, uppercase k key, right parentheses, Returns true if key is a key in this Hash Map, false otherwise. Row 8. Return value: boolean. Method name and argument list: contains, left parentheses, uppercase v value, right parentheses, Returns true if value is mapped to one or more keys in this Hash Map, false otherwise. Row 9. Return value: Set, left angled bracket, uppercase k, right angled bracket. Method name and argument list: key Set, left parentheses, right parentheses, Returns a Set of keys contained in this Hash Map. Row 10. Return value: Collection, left angled bracket, uppercase v, right angled bracket. Method name and argument list: values, left parentheses, right parentheses, Returns a collection of values contained in this Hash Map.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, String, for. Line 1: Forward slash. Asterisk. A simple Hash Map of, left parentheses, String, comma, String, right parentheses, pairs. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Hash Map, semicolon. Line 6: import java dot util dot Set, semicolon. Line 7: blank. Line 8: public class Hash Map States. Line 9: left brace. Line 10, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: Hash Map, left angled bracket, String, comma, String, right angled bracket, states, equals, new Hash Map, left angled bracket, String, comma, String, right angled bracket, left parentheses, right parentheses, semicolon. Line 13, indented twice: states dot put, left parentheses, double quotes, C A, double quotes, comma, double quotes, California, double quotes, right parentheses, semicolon. Line 14, indented twice: states dot put, left parentheses, double quotes, M D, double quotes, comma, double quotes, Maine, double quotes, right parentheses, semicolon. 2 forward slashes, We correct this later. Line 15, indented twice: states dot put, left parentheses, double quotes, N Y, double quotes, comma, double quotes, New York, double quotes, right parentheses, semicolon. Line 16, indented twice: states dot put, left parentheses, double quotes, N J, double quotes, comma, double quotes, New Jersey, double quotes, right parentheses, semicolon. Line 17: blank. Line 18, indented twice: 2 forward slashes, Output all, left parentheses, key, comma, value, right parentheses, pairs. Line 19, indented twice: System dot out dot print l n, left parentheses, double quotes, states, colon, blank space, double quotes, plus, states, right parentheses, semicolon. Line 20: blank. Line 21, indented twice: 2 forward slashes, Retrieve value whose key is N Y. Line 22, indented twice: String n y State, equals, states dot get, left parentheses, double quotes, N Y, double quotes, right parentheses, semicolon. Line 23, indented twice: System dot out dot print l n, left parentheses, double quotes, N Y is, double quotes, plus, n y State, right parentheses, semicolon. Line 24: blank. Line 25, indented twice: 2 forward slashes. Change value whose key is M D to Maryland. Line 26, indented twice:

states dot replace, left parentheses, double quotes, M D, double quotes, comma, double quotes, Maryland, double quotes, right parentheses, semicolon. Line 27, indented twice: System dot out dot print l n, left parentheses, double quotes, states, colon, blank space, double quotes, plus, states, plus, double quotes, backslash, n, double quotes, right parentheses, semicolon. Line 28: blank. Line 29, indented twice: 2 forward slashes. Remove state whose key is M D. Line 30, indented twice: String m d State, equals, states dot remove, left parentheses, double quotes, M D, double quotes, right parentheses, semicolon. Line 31, indented twice: System dot out dot print l n, left parentheses, m d State, plus, double quotes, was removed, double quotes, right parentheses, semicolon. Line 32, indented twice: System dot out dot print l n, left parentheses, double quotes, states, colon, double quotes, plus, states, plus, double quotes, backslash, n, double quotes, right parentheses, semicolon. Line 33: blank. Line 34, indented twice: 2 forward slashes. Loop through all the values using the keys. Line 35, indented twice: Set, left angled bracket, String, right angled bracket, keys, equals, states, dot, key Set, left parentheses, right parentheses, semicolon. Line 36, indented twice: for, left parentheses, String a b b r, colon, keys, right parentheses. Line 37, indented thrice: System dot out dot print l n, left parentheses, a b b r, plus, double quotes, colon, double quotes, plus, states dot get, left parentheses, a b b r, right parentheses, right parentheses, semicolon. Line 38, indented twice: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 39: blank. Line 40, indented twice: 2 forward slashes, Loop through all the values directly. Line 41, indented twice: for, left parentheses, String state, colon, states dot values, left parentheses, right parentheses, right parentheses. Line 42, indented thrice: System dot out dot print l n, left parentheses, state, right parentheses, semicolon. Line 43, indented once: right brace. Line 44: right brace.

Back to Example

Line 1: String, left square bracket, right square bracket, left square bracket, right square bracket, cities, equals, left brace, left brace, double quotes, New York, double quotes, comma, double quotes, uppercase l, uppercase a, double quotes, comma, double quotes, San Francisco, double quotes, comma, double quotes, Chicago, double quotes, right brace, comma. Line 2, indented multiple times: left brace, double quotes, Munich, double quotes, comma, double quotes, Stuttgart, double quotes, comma, double quotes, Berlin, double quotes, comma, double quotes, Bonn, double quotes, right brace, comma. Line 3, indented multiple times: left brace, double quotes, Paris, double quotes, comma, double quotes, Ajaccio, double quotes, comma, double quotes, Lyon, double quotes, right brace, comma. Line 4, indented multiple times: left brace, double quotes, Montreal, double quotes, comma, double quotes, Ottawa, double quotes, comma, double quotes, Vancouver, double quotes, right brace, right brace,

semicolon.

Back to Code

System dot out dot print l n, left parentheses, cities, left square bracket, 3, right square bracket, left square bracket, 2, right square bracket, right parentheses, semicolon.

Back to Code

Line 1: for, left parentheses, int j equals 0, semicolon, j less than cities, left square bracket, i, right square bracket, dot length, semicolon, j plus plus, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, cities, left square bracket, i, right square bracket, left square bracket, j, right square bracket, right parentheses, semicolon.

Back to Code

Line 1: for, left parentheses, int i equals 0, semicolon, i less than cities dot length, semicolon, i plus, plus, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, cities, left square bracket, i, right square bracket, left square bracket, 1, right square bracket, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i less than cities dot length, semicolon, i plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int j equals 0, semicolon, j less than cities, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 4, indented twice: System dot out dot print, left parentheses, cities, left square bracket, i, right square bracket, left square bracket, j, right square bracket, plus, double quotes, back slash, t, double quotes, right parentheses, semicolon. Line 5, indented once: System dot out dot print I n, left parentheses, right parentheses, semicolon. Line 6: right brace.

Back to Code

Program code. The code contains the following keywords: for, int, if. Line 1: for, left parentheses, int i equals 0, semicolon, i less than cities dot length, semicolon, i, plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int j equals 0, semicolon, j less than cities, left square bracket, i, right square bracket, dot length, semicolon, j plus plus, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: if, left parentheses, cities, left square bracket, i, right square bracket, left square bracket, j, right square bracket, dot length, left parentheses, right parentheses, equals, equals, 6, right parentheses. Line 6, indented thrice: System dot out dot print I n, left parentheses, cities, left square bracket, i, right square bracket, left square bracket, j, right square bracket, right parentheses, semicolon. Line 7, indented once: right brace. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: int, for, if. Line 1: int count equals 0, semicolon. Line 2: for, left parentheses, int i equals 0, semicolon, i less than cities dot length, semicolon, i, plus, plus, right parentheses. Line 3: left brace. Line 4, indented once: for, left parentheses, int j equals 0, semicolon, j less than cities, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: if, left parentheses, cities, left square bracket, i, right square bracket, left square bracket, j, right square bracket, dot, length, left parentheses, right parentheses, equals, equals, 7, right parentheses. Line 7, indented thrice: count, plus, plus, semicolon. Line 8, indented once: right brace. Line 9: right brace.

Back to Code

Program code. The code contains the following keywords: for, int, if. Line 1: for, left parentheses, int i equals 0, semicolon, i less than cities dot length, semicolon, i, plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int j equals 0, semicolon, j less than cities, left square bracket, i, right square bracket, dot, length, semicolon, j, plus, plus, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: if, left parentheses, cities, left square bracket, i, right square bracket, left square bracket, j, right square bracket, dot, char At, left parentheses, 0, right parentheses, equals, equals, single quote, uppercase s, single quote, right parentheses. Line 6, indented thrice: System dot out dot print l n, left parentheses, cities, left square bracket, i, right square bracket, left square bracket, j, right square bracket, right parentheses, semicolon. Line 7, indented once: right brace. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, int, double, for, return. Line 1: public static int foo, left parentheses, double, left square bracket, right square bracket, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: int b equals 0, semicolon. Line 4, indented once: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: for, left parentheses, int j equals 0, semicolon, j less than a, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 7, indented thrice: b plus, plus, semicolon. Line 8, indented once: right brace. Line 9, indented once: return b, semicolon. Line 10: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, boolean, char, int. Line 1: public static boolean foo, left parentheses, char, left square bracket, right square bracket, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: int b equals a, left square bracket, 0, right square bracket, dot, length, semicolon.

Back to Code

Program code. The code contains the following keywords: public, static, int, String, for, return. Line 1: public static int foo, left parentheses, String, left square bracket, right square bracket, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: int b equals 0, semicolon. Line 4, indented once: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i plus, plus, right parentheses. Line 5, indented twice: b plus, plus, semicolon. Line 6, indented once: return b, semicolon. Line 7: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, int, float, new, for, return. Line 1: public static int, left square bracket, right square bracket, foo, left parentheses, float, left square bracket, right square bracket, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: int, left square bracket, right square bracket, temp, equals, new int, left square bracket, a dot length, right square bracket, semicolon. Line 4, indented once: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i plus, plus, right parentheses. Line 5, indented twice: temp, left square bracket, i, right square bracket, equals, a, left square bracket, i, right square bracket, dot, length, semicolon. Line 6, indented once: return temp, semicolon. Line 7: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, int, for, return. Line 1: public static int foo, left parentheses, Array List, left angled bracket, Integer, right angled bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: int b equals 0, semicolon. Line 4, indented once: for, left parentheses, Integer i, colon, a, right parentheses. Line 5, indented twice: b plus, plus, semicolon. Line 6, indented once: return b, semicolon. Line 7: right brace.

Back to Code

Line 1: Array List, left angled bracket, Integer, right angled bracket, a, equals, new Array List, left angled bracket, Integer, right angled bracket, left parentheses, semicolon. Line 2: a dot add, left parentheses, 7, right parentheses, semicolon. Line 3: a dot add, left parentheses, 4, right parentheses, semicolon. Line 4: a dot add, left parentheses, 21, right parentheses, semicolon.

Back to Code

Line 1: Array List, left angled bracket, Integer, right angled bracket, a, equals, new Array List, left angled bracket, Integer, right angled bracket, left parentheses, semicolon. Line 2: a dot add, left parentheses, 7, right parentheses, semicolon. Line 3: a dot add, left parentheses, 4, right parentheses, semicolon. Line 4: a dot add, left parentheses, 21, right parentheses, semicolon. Line 5: a dot set, left parentheses, 1, comma, 45, right parentheses, semicolon.

Back to Code

Line 1: Hash Map, left angled bracket, String, comma, String, right angled bracket, days, equals, new Hash Map, left angled bracket, String, comma, String, right angled bracket, left parentheses, right parentheses, semicolon. Line 2: days dot put, left parentheses, double quotes, SUN, double quotes, comma, double quotes, Sunday, double quotes, right parentheses, semicolon. Line 3: days dot put, left parentheses, double quotes, M O N, double quotes, comma, double quotes, Monday, double quotes, right parentheses, semicolon. Line 4: days dot put, left parentheses, double quotes, T U E S, double quotes, comma, double quotes, Tuesday, double quotes, right parentheses, semicolon.

Back to Code

Line 1: String, left square bracket, right square bracket, left square bracket, right square bracket, g e o, equals, left brace, left brace, double quotes, M D, double quotes, comma, double quotes, N Y, double quotes, comma, double quotes, N J, double quotes, comma, double quotes, M A, double quotes, comma, double quotes, M E, double quotes, comma, double quotes, C A, double quotes, comma, double quotes, M I, double quotes, comma, double quotes, O R, double quotes, right brace, comma. Line 2, indented multiple times: left brace, double quotes, Detroit, double quotes, comma, double quotes, Newark, double quotes, comma, double quotes, Boston, double quotes, comma, double quotes, Seattle, double quotes, right brace, right brace, semicolon.

Back to Code

Program code. The code contains the keywords for and int. Line 1: for, left parentheses, int j equals 0, semicolon, j less than geo, left square bracket, 0, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, your code goes here. Line 4: right brace.

Back to Code

Program code. The code contains the keywords for and int. Line 1: for, left parentheses, int j equals 0, semicolon, j less than geo, left square bracket, 1, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, your code goes here. Line 4: right brace.

Back to Code

Program code. The code contains the keywords for and int. Line 1: for, left parentheses, int i equals 0, semicolon, j less than geo, dot, length, semicolon, i plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, your code goes here. Line 4: right brace.

Back to Code

Line 1: int, left square bracket, right square bracket, left square bracket, right square bracket, a, equals, left brace, left brace, 9, comma, 6, comma, 8, comma, 10, comma, 5, right brace, comma. Line 2, indented multiple times: left brace, 7, comma, 6, comma, 8, comma, 9, comma, 6, right brace, comma. Line 3, indented multiple times: left brace, 4, comma, 8, comma, 10, comma, 6, comma, 6, right brace, right brace, semicolon.

Back to Code

Program code. The code contains the keywords int and for. Line 1: int sum equals 0, semicolon. Line 2: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i plus, plus, right parentheses. Line 3: left brace. Line 4, indented once: 2 forward slashes, your code goes here. Line 5: right brace. Line 6: System dot out dot print l n, left parentheses, double quotes, sum is, double quotes, plus, sum, right parentheses, semicolon.

Back to Code

Program code. The code contains the keywords int and for. Line 1: int count equals 0, semicolon. Line 2: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i plus, plus, right parentheses. Line 3: left brace. Line 4, indented once: 2 forward slashes, your code goes here. Line 5: right brace. Line 6: System dot out dot print l n, left parentheses, double quotes, hash symbol, of 8 s in a, colon, blank space, double quotes, plus, count, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: int, if. Line 1: int count equals 0, semicolon. Line 2: blank. Line 3: 2 forward slashes, your code for the loop header goes here. Line 4: left brace. Line 5, indented once: if, left parentheses, a, left square bracket, 1, right square bracket, left square bracket, j, right square bracket, equals, equals, 6, right parentheses. Line 6, indented twice: count, plus, plus, semicolon. Line 7: right brace. Line 8: System dot out dot print l n, left parentheses, double quotes, hash symbol of 6 s in the 2 n d row, colon, blank space, double quotes, plus, count, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: int, for. Line 1: int sum equals 0, semicolon. Line 2: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i plus, plus, right parentheses. Line 3: left brace. Line 4, indented once: 2 forward slashes, your code goes here. Line 5: right brace. Line 6: System dot out dot print l n, left parentheses, double quotes, sum is, blank space, double quotes, plus, sum, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, static, boolean, String. Line 1: public static boolean foo, left parentheses, String, left square bracket, right square bracket, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, your code goes here. Line 4: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, int. Line 1: public static int foo, left parentheses, int, left square bracket, right square bracket, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, your code goes here. Line 4: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, boolean, char. Line 1: public static boolean foo, left parentheses, char, left square bracket, right square bracket, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, your code goes here. Line 4: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, String. Line 1: public static Array List, left angled bracket, String, right angled bracket, foo, left parentheses, Hash Map, left angled bracket, String, comma, String, right angled bracket, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, your code goes here. Line 4: right brace.

Back to Code

Line 1: Array List, left angled bracket, String, right angled bracket, languages, equals, new Array List, left angled bracket, String, right angled bracket, left parentheses, right parentheses, semicolon. Line 2: languages dot add, left parentheses, double quotes, S Q L, double quotes, right parentheses, semicolon. Line 3: languages dot add, left parentheses, double quotes, Java, double quotes, right parentheses, semicolon. Line 4: languages dot add, left parentheses, double quotes, H T M L, double quotes, right parentheses, semicolon. Line 5: languages dot add, left parentheses, double quotes, P H P, double quotes, right parentheses, semicolon. Line 6: languages dot add, left parentheses, double quotes, Perl, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the keywords for and String. Line 1: for, left parentheses, String, s, colon, languages, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, your code goes here. Line 4: right brace.

Back to Code

double, left square bracket, right square bracket, left square bracket, right square bracket, a, equals, left brace, 3.3, comma, 26.0, comma, 48.4, right brace, semicolon.

Back to Code

Line 1: int, left square bracket, right square bracket, left square bracket, right square bracket, a, equals, left brace, left brace, 3, comma, 26, comma, 4, right brace, comma, left brace, 14, comma, 87, right brace, right brace, semicolon. Line 2: System dot out dot print l n, left parentheses, a, left square bracket, 1, right square bracket, left square bracket, 2, right square bracket, right parentheses, semicolon.

Back to Code

double, left square bracket, right square bracket, left square bracket, right square bracket, a, equals, new double, left square bracket, right square bracket, left square bracket, 10, right square bracket, semicolon.

Back to Code

Line 1: int, left square bracket, right square bracket, left square bracket, right square bracket, a, equals, left brace, left brace, 1, comma, 2, right brace, comma. Line 2, indented once: left brace, 10.1, comma, 10.2, right brace, right brace, semicolon.

Back to Code

Line 1: int, left square bracket, right square bracket, left square bracket, right square bracket, a, equals, left brace, left brace, 3, comma, 26, comma, 48, right brace, comma, left brace, 5, comma, 2, comma, 9, right brace, right brace, semicolon. Line 2: System dot out dot print l n, left brace, double quotes, The array elements are, blank space, double quotes, plus, a, right parentheses, semicolon.

Back to Code

Array List, left angled bracket, double, right angled bracket, a l, semicolon.

Back to Code

Array List, left angled bracket, Float, right angled bracket, a 1, equals, new Array List, left parentheses, right parentheses, left angled bracket, Float, right angled bracket, semicolon.

Back to Code

Line 1: Array List, left angled bracket, Double, right angled bracket, a, semicolon. Line 2: a equals new Array List, left angled bracket, Float, right angled bracket, left parentheses, right parentheses, semicolon.

Back to Code

Line 1: 2 forward slashes, a is an Array List of Strings. Line 2: 2 forward slashes, a has already been declared and instantiated. Line 3: a dot size, left parentheses, right parentheses, equals, 10, semicolon.

Back to Code

int a, left square bracket, 2, right square bracket, left square bracket, right square bracket, equals, left brace, left brace, 2, comma, 7, right brace, comma, left brace, 9, comma, 2, right brace, right brace, semicolon. 2 forward slashes, line 14.

Back to Code

Line 1: Test dot java, colon, 14, colon, error, colon, single quote, right angled bracket, single quote, expected. Line 2, indented once: int a, left square bracket, 2, right square bracket, left square bracket, right square bracket, equals, left brace, left brace, 2, comma, 7, right brace, comma, left brace, 9, comma, 2, right brace, right brace, semicolon. 2 forward slashes, line 14. Line 3: a caret sign under the first occurence of character 2 in Line 2. Line 4: Test dot java, colon, 14, colon, error, colon, not a statement. Line 5, indented once: int, a, left square bracket, 2, right square bracket, left square bracket, right square bracket, equals, left brace, left brace, 2, comma, 7, right brace, comma, left brace, 9, comma, 2, right brace, right brace, semicolon. 2 forward slashes, line 14. Line 6: a caret sign under the second occurence of character 2 in Line 5. Line 7: Test dot java, colon, 14, colon, error, colon, single quote, semicolon, single quote, expected. Line 8, indented once: int a, left square bracket, 2, right square bracket, left square bracket, right square bracket, equals, left brace, left brace, 2, comma, 7, right brace, comma, left brace, 9, comma, 2, right brace, right brace, semicolon. 2 forward slashes, line 14. Line 9: A caret sign under the comma following the second occurence of character 2 in Line 8. Line 10: Test dot java, colon, 14, colon, error, colon, illegal start of expression. Line 11, indented once: int a, left square bracket, 2, right square bracket, left square bracket, right square bracket, equals, left brace, left brace, 2, comma, 7, right brace, comma, left brace, 9, comma, 2, right brace, right brace, semicolon. 2 forward slashes, line 14. Line 12: a caret sign under the comma following the first right brace character. Line 13: Test dot java, colon, 14, colon, error, colon, not a statement. Line 14, indented once: int a, left square bracket, 2, right square bracket, left square bracket, right square bracket, equals, left brace, left brace, 2, comma, 7, right brace, comma, left brace, 9, comma, 2, right brace, right brace, semicolon. 2 forward slashes, line 14. Line 15: A caret sign under the character 9 of Line 14. Line 15: Test dot java, colon, 14, colon, error, colon, single quote, semicolon, single quote, expected. Line 16, indented once: int, left square bracket, 2, right square bracket, left square bracket, right square bracket, equals, left brace, left brace, 2, comma, 7, right brace, comma, left brace, 9, comma, 2, right brace, right brace, semicolon. 2 forward slashes, line 14. Line 17: A caret sign under the comma following the character 9 in Line 16. Line 18: 6 errors.

Back to Code

Program code. The code contains the following keywords: int, for. Line 1: int, left square bracket, right square bracket, left square bracket, right square bracket, a, equals, left brace, left brace, 1, comma, 2, comma, 3, comma, 4, right brace, comma. Line 2, indented multiple times: left brace, 10, comma, 20, comma, 30, right brace, right brace, semicolon. Line 3: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i plus, plus, right parentheses. Line 4: left brace. Line 5, indented once: for, left parentheses, int j equals 0, semicolon, j less than a, left square bracket, 0, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 6, indented once: left brace. Line 7, indented twice: System dot out dot print l n, left parentheses, a, left square bracket, i, right square bracket, left square bracket, j, right square bracket, right parentheses, semicolon. 2 forward slashes, line 14. Line 8, indented once: right brace. Line 9: right brace.

Back to Code

Line 1: 1. Line 2: 2. Line 3: 3. Line 4: 4. Line 5: 10. Line 6: 20. Line 7: 30. Line 8: Exception in thread, double quotes, main, double quotes, java dot lang dot Array Index Out Of Bounds Exception, colon. Line 9, indented once: Index 3 out of bounds for length 3. Line 10, indented once: at Test dot main, left parentheses, Test dot java, colon, 14, right parentheses.

Back to Code

Program code. The code contains the following keywords: int, for, if. Line 1: int, left square bracket, right square bracket, left square bracket, right square bracket, a, equals, left brace, left brace, 9, comma, 8, comma, 7, comma, 6, right brace, comma. Line 2, indented multiple times: left brace, 10, comma, 20, comma, 30, comma, 40, right brace, right brace, semicolon. Line 3: int min equals a, left square bracket, 0, right square bracket, left square bracket, 0, right square bracket, semicolon. Line 4: for, left parentheses, int i equals 1, semicolon, i less than a dot length, semicolon, i plus, plus, right parentheses. Line 5: left brace. Line 6, indented once: for, left parentheses, int j equals 0, semicolon, j less than a, left square bracket, i, right square bracket, dot length, semicolon, j plus, plus, right parentheses. Line 7, indented once: left brace. Line 8, indented twice: if, left parentheses, a, left square bracket, i, right square bracket, left square bracket, j, right square bracket, less than, min, right parentheses. Line 9, indented thrice: min equals a, left square bracket, i, right square bracket, left square bracket, j, right square bracket, semicolon. Line 10, indented once: right brace. Line 11: right brace. Line 12: System dot out dot print l n, left parentheses, double quotes, The minimum is, blank space, double quotes, plus, min, right parentheses, semicolon.

Back to Code

The minimum is 9.

Back to Code

Program code. The code contains the following keywords: int, for, if. Line 1: int, left square bracket, right square bracket, left square bracket, right square bracket, a, equals, left brace, left brace, 9, comma, 8, comma, 7, comma, 6, right brace, comma. Line 2, indented multiple times: left brace, 10, comma, 20, comma, 30, comma, 40, right brace, right brace, semicolon. Line 3: for, left parentheses, int j equals 0, semicolon, j less than or equal to a, left square bracket, 1, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. Line 4: left brace. Line 5, indented once: if, left parentheses, a, left square bracket, 1, right square bracket, left square bracket, j, right square bracket, equals, equals, 20, right parentheses, 2 forward slashes, line 14. Line 6, indented once: left brace. Line 7, indented twice: System dot out dot print l n, left parentheses, double quotes, Found 20 at column index, blank space, double quotes, plus j. Line 8, indented multiple times: plus, double quotes, of second row, double quotes, right parentheses, semicolon. Line 9, indented once: right brace. Line 10: right brace.

Back to Code

Line 1: Found 20 at column index 1 of second row. Line 2: Exception in thread, double quotes, main, double quotes, java dot lang dot Array Index Out Of Bounds Exception, colon, Index 4. Line 3, indented once: out of bounds for length 4. Line 4, indented once: at Test dot main, left parentheses, Test dot java, colon, 14, right parentheses.

Back to Code

Program code. The code contains the following keywords: for, Auto. Line 1: 2 forward slashes, cars is an Array List of Auto objects. Line 2: 2 forward slashes, cars has already been declared and instantiated. Line 3: for, left parentheses, Auto a, blank space, semicolon, cars, right parentheses, blank spaces, 2 forward slashes, line 12. Line 4: left brace. Line 5, indented once: System dot out dot print l n, left parentheses, a dot to String, left parentheses, right parentheses, right parentheses, semicolon. Line 6: right brace, 2 forward slashes, line 15.

Back to Code

Line 1: Test dot java, colon, 12, colon, error, colon, single quote, semicolon, single quote, expected. Line 2, indented once: for, left parentheses, Auto a, semicolon, cars, blank space, right parentheses, 2 forward slashes, line 12. Line 3: a caret sign under the blank space before the right parentheses in Line 2. Line 4: 1 error.

Back to Code

Line 1: Array List, left angled bracket, String, right angled bracket, a, equals, new, Array List, let angled bracket, String, right angled bracket, left parentheses, right parentheses, semicolon. Line 2: a dot add, left parentheses, double quotes, Cloudy, double quotes, right parentheses, semicolon. Line 3: a dot add, left parentheses, double quotes, Snowy, double quotes, right parentheses, semicolon. Line 4: a dot add, left parentheses, double quotes, Cloudy, double quotes, right parentheses, semicolon.

Back to Code

Line 1: Exception in thread, double quotes, main, double quotes, java dot Lang dot Index Out Of Bounds Exception, colon, Index 3 out of. Line 2: bounds for length 3. Line 3: dot, dot, dot.

Back to Code

Array List, left angled bracket, Integer, right angled bracket, a, equals, new Array List, left parentheses, right parentheses, semicolon.

Back to Code

Line 1: Test dot java, colon, 10, colon, warning, colon, left square bracket, raw types, right angled bracket, found raw type, colon, Array List. Line 2, indented once: Array List, left angled bracket, Integer, right angled bracket, a, equals, new Array List, left parentheses, right parentheses, semicolon. Line 3: a caret sign under the uppercase A following new in Line 2. Line 4, indented once: missing type arguments for generic class Array List, left angled bracket, uppercase e, right angled bracket. Line 5, indented once: where uppercase e is a type hyphen variable, colon. Line 6, indented once: uppercase e extends Object declared in class Array List. Line 7: blank. Line 8: Test dot java, colon, 10, colon, warning, colon, left square bracket, unchecked, right square bracket, unchecked conversion. Line 9, indented once: Array List, left angled bracket, Integer, right angled bracket, a, equals, new Array List, left parentheses, right parentheses, semicolon. Line 10: a caret sign under lowercase n of new in Line 9. Line 11, indented once: required, colon, Array List, left angled bracket, Integer, right angled bracket. Line 12, indented once: found, colon, Array List. Line 13: 2 warnings.

Back to Code

Line 1: Array List, left angled bracket, Double, right angled bracket, a, equals, new Array List, left angled bracket, Double, right angled bracket, left parentheses, right parentheses, semicolon. Line 2: a dot add, left parentheses, 2.3, right parentheses, semicolon. Line 3: a dot add, left parentheses, 8.4, right parentheses, semicolon. Line 4: a dot add, left parentheses, 5, right parentheses, semicolon. 2 forward slashes, line 11.

Back to Code

Line 1: Test dot java, colon, 11, colon, error, colon, incompatible types, colon, int cannot be converted to Double. Line 2, indented once: a dot add, left parentheses, 5, right parentheses, semicolon. 2 forward slashes, line 11. Line 3: a caret sign under 5 of Line 2. Line 3: Note, colon, Some messages have been simplified, semicolon, recompile with hyphen, uppercase x, d i a g s, colon, verbose. Line 4, indented once: to get full output. Line 5: 1 error. Line 6: a caret sign under 1 in Line 5.

Back to Code

Line 1: Array List, left angled bracket, Character, right angled bracket, a, equals, new Array List, left angled bracket, Character, right angled bracket, left parentheses, right parentheses, semicolon. Line 2: a dot add, left parentheses, single quote, uppercase x, single quote, right parentheses, semicolon. Line 3: a dot add, left parentheses, single quote, uppercase a, single quote, right parentheses, semicolon. Line 4: a dot add, left parentheses, single quote, uppercase v, single quote, right parentheses, semicolon. Line 5: a dot add, left parentheses, single quote, uppercase a, single quote, right parentheses, semicolon. Line 6: a dot set, left parentheses, 1, comma, single quote, uppercase j, single quote, right parentheses, semicolon. Line 7: for, left parentheses, Character, c, colon, a, right parentheses. Line 8, indented once: System dot out dot print, left parentheses, c plus, double quotes, blank space, double quotes, right parentheses, semicolon.

Back to Code

The text magnified by the lens reads: Look It Up.

Back to Figure

Program code. The code contains the following keywords: import, public, class, final, new, private, double, return, if, this, String. Line 1: Forward slash, 2 asterisks. Bank Account class, comma, version 1. Line 2: Asterisk. Tab space. Anderson, comma, Franceschi. Line 3: Asterisk. Tab space. Represents a generic bank account. Line 4: Asterisk, forward slash. Line 5: import java dot text dot Decimal Format, semicolon. Line 6: blank. Line 7: public class Bank Account. Line 8: left brace. Line 9, indented once: public final Decimal Format MONEY. Line 10, indented multiple times: equals, new, Decimal Format, left parentheses, double quotes, dollar sign, hash, comma, 2 hashes, 0, decimal point, 0 0, double quotes, right parentheses, semicolon. Line 11, indented once: private double balance, semicolon. Line 12: blank. Line 13, indented once: Forward slash, 2 asterisks, default constructor. Line 14, indented once: asterisk, blank space, sets balance to 0, decimal point, 0. Line 15, indented once: Asterisk, forward slash. Line 16, indented once: public Bank Account, left parentheses, right parentheses. Line 17, indented once: left brace. Line 18, indented twice: balance, equals, 0.0, semicolon. Line 19, indented once: right brace. Line 20: blank. Line 21, indented once: Forward slash, 2 asterisks, overloaded constructor. Line 22, indented once: asterisk, blank spaces, at symbol, param balance begining balance. Line 23, indented once: asterisk, forward slash. Line 24, indented once: public Bank Account, left parentheses, double balance, right parentheses. Line 25, indented once: left brace. Line 26, indented twice: deposit, left parentheses, balance, right parentheses, semicolon. Line 27, indented once: right brace. Line 28: blank. Line 29, indented once: Forward slash, 2 asterisks, accessor for balance. Line 30, indented once: asterisk, blank spaces, at symbol, return current account balance. Line 31, indented once: Asterisk, forward slash. Line 32, indented once: public double get Balance, left parentheses, right parentheses. Line 33, indented once: left brace. Line 34, indented twice: return balance, semicolon. Line 35, indented once: right brace. Line 36: blank. Line 37, indented once: Forward slash, 2 asterisks, deposit amount to account. Line 38, indented once: Asterisk, blank spaces, at symbol, param amount amount to deposit, semicolon. Line 39, indented once: Asterisk, blank spaces, amount must be, greater than or equal to 0.0. Line 40, indented once: Asterisk, blank spaces, at symbol, return a reference to this object. Line 41, indented once: Asterisk, forward slash. Line 42, indented once: public Bank Account deposit, left parentheses, double amount, right parentheses. Line 43, indented once: left brace. Line 44, indented twice: if, left parentheses, amount greater than or equal to 0.0, right parentheses. Line 45, indented thrice: balance, plus, equals, amount, semicolon. Line 46: blank. Line 47, indented twice: return this, semicolon. Line 48, indented once: right brace. Line 49: blank. Line 50, indented once: Forward slash, 2 asterisks. withdraw amount from account. Line 51, indented once: asterisk, blank spaces, at symbol, param amount, blank spaces, amount to withdraw, semicolon. Line 52, indented once: Asterisk, blank spaces, amount must be greater than or equal to 0.0. Line 53, indented once: Asterisk, blank spaces, amount must be less than or equal to balance. Line 54, indented once: asterisk, blank spaces, at symbol, return a reference to this object. Line 55: Asterisk. Forward slash. Line 56, indented once: public Bank Account withdraw, left parentheses, double amount, right parentheses. Line 56, indented once: left brace. Line 58, indented twice: if, left parentheses, amount greater than or equal to 0.0, 2 ampersands, amount less than or equal to balance, right parentheses. Line 59, indented thrice: balance, minus, equals, amount, semicolon. Line 60: blank. Line 61, indented twice: return this, semicolon. Line 62, indented once: right brace. Line 63: blank. Line 64, indented once: Forward slash, 2 asterisks, to String. Line 65, indented once: Asterisk, blank spaces, at symbol, return the balance formatted as money. Line 66, indented once: Asterisk. Forward slash. Line 67, indented once: at symbol, Override. Line 68, indented once: public String to String, left parentheses, right parentheses. Line 69, indented once: left brace. Line 70, indented twice: return, double quotes, balance is, double quotes, plus, MONEY dot format, left parentheses, balance, right parentheses, semicolon. Line 71, indented once: right brace. Line 72: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, extends. Line 1: Forward slash, asterisk. Checking Account class, comma, Version 1. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Checking Account extends Bank Account. Line 6: left brace, right brace.

Back to Example

The table has 4 columns labeled Superclass Members, Inherited by Subclass, question mark, Directly Accessible by Subclass, question mark, Directly Accessible by Client of Subclass, question mark. The row entries are as follows. Row 1. Superclass Members: public fields. Inherited by Subclass, question mark: yes. Directly Accessible by Subclass, question mark: yes, by using field name. Directly Accessible by Client of Subclass, question mark: yes. Row 2. Superclass Members: public methods. Inherited by Subclass, question mark: yes. Directly Accessible by Subclass, question mark: yes, by calling method from other subclass methods. Directly Accessible by Client of Subclass, question mark: yes. Row 3. Superclass Members: protected fields. Inherited by Subclass, question mark: yes. Directly Accessible by Subclass, question mark: yes, by using field name. Directly Accessible by Client of Subclass, question mark: no, must use accessors and mutators. Row 4. Superclass Members: protected methods. Inherited by Subclass, question mark: yes. Directly Accessible by Subclass, question mark: yes, by calling method from subclass methods. Directly Accessible by Client of Subclass, question mark: no. Row 5. Superclass Members: private fields. Inherited by Subclass, question mark: no. Directly Accessible by Subclass, question mark: no, must use accessors and mutators. Directly Accessible by Client of Subclass, question mark: no, must use accessors and mutators. Row 6. Superclass Members: private methods. Inherited by Subclass, question mark: no. Directly Accessible by Subclass, question mark: no. Directly

Accessible by Client of Subclass, question mark: no.

Back to Table

Program code. The code contains the following keywords: public, class, static, void. Line 1: Forward slash, asterisk. Checking Account Client, comma, Version 1. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Checking Account Client. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: Checking Account c 1, equals, new Checking Account, left parentheses, right parentheses, semicolon. Line 10, indented twice: System dot out dot print l n, left parentheses, double quotes, New checking account, colon, blank space, double quotes, plus, c 1, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: c 1 dot deposit, left parentheses, 350.75, right parentheses, semicolon. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, New checking account, colon, blank space, double quotes, plus c 1, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: c 1 dot withdraw, left parentheses, 200.25, right parentheses, semicolon. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash n, After withdrawing dollar sign, 200.25, colon, blank space, double quotes, plus, c 1, right parentheses, semicolon.

Line 17, indented once: right brace. Line 18: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, final, new, double, private, if, this, else, return, at symbol Override. Line 1: Forward slash. 2 asterisks. Bank Account class, comma, Version 2. Line 2: Asterisk, tab space. Constructors and to String method only. Line 3: Asterisk. Tab space. Anderson, comma, Franceschi. Line 4: Asterisk. Tab space. Represents a generic bank account. Line 5: Asterisk. Forward slash. Line 6: blank. Line 7: import java dot text dot Decimal Format, semicolon. Line 8: blank. Line 9: public class Bank Account. Line 10: left brace. Line 11, indented once: public final Decimal Format MONEY. Line 12, indented multiple times: equals, new Decimal Format, left parentheses, double quotes, dollar sign, hash, comma, 2 hashes, 0, decimal point, 0 0, double quotes, right parentheses, semicolon. Line 13, indented once: private double balance, semicolon. Line 14: blank. Line 15, indented once: Forward slash, 2 asterisks. default constructor. Line 16, indented once: Asterisk. Tab space. sets balance to 0.0. Line 17, indented once: Asterisk. Forward slash. Line 18, indented once: public Bank Account, left parentheses, right parentheses. Line 19, indented once: left brace. Line 20, indented twice: balance, equals, 0.0, semicolon. Line 21, indented twice: System dot out dot print l n, left parentheses, double quotes, In Bank Account default constructor, double quotes, right parentheses, semicolon. Line 22, indented once: right brace. Line 23: blank. Line 24, indented once: Forward slash. 2 asterisks. overloaded constructor. Line 25, indented once: Asterisk. Blank spaces. at symbol, param balance begining balance. Line 26, indented once: Asterisk, forward slash. Line 27, indented once: public Bank Account, left parentheses, double balance, right parentheses. Line 28, indented once: left brace. Line 29, indented twice: if, left parentheses, balance, greater than or equal to, 0.0, right parentheses. Line 30, indented thrice: this dot balance, equals, balance, semicolon. Line 31, indented twice: else. Line 32, indented thrice: this dot balance, equals, 0.0, semicolon. Line 33, indented twice: System dot out dot print l n, left parentheses, double quotes, In Bank Account overloaded constructor, double quotes, right parentheses, semicolon. Line 34, indented once: right brace. Line 35: blank. Line 36, indented once: Forward slash. 2 asterisks. to String. Line 37, indented once: Asterisk. Blank spaces. at symbol, return the balance formatted as money. Line 38, indented once: Asterisk. Forward slash. Line 39, indented once: at symbol, Override. Line 40, indented once: public String to String, left parentheses, right parentheses. Line 41, indented once: left brace. Line 42, indented twice: return double quotes, balance is, blank space, double quotes, plus, MONEY dot format, left parentheses, balance, right parentheses, semicolon. Line 43, indented once: right brace. Line 44: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, extends, double. Line 1: Forward slash. Asterisk. Checking Account class, comma, version 2. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Checking Account extends Bank Account. Line 6: left brace. Line 7, indented once: Forward slash. 2 asterisks. default constructor. Line 8, indented once: Asterisk, blank spaces. explicitly calls the Bank Account default constructor. Line 9, indented once: Asterisk. Forward slash. Line 10, indented once: public Checking Account, left parentheses, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: super, left parentheses, right parentheses, semicolon. 2 forward slashes. optional, comma, call Bank Account constructor. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, In Checking Account, blank space, double quotes. Line 14, indented multiple times: plus, double quotes, default constructor, double quotes, right parentheses, semicolon. Line 15, indented: right brace. Line 16: blank. Line 17, indented once: Forward slash. 2 asterisks. overloaded constructor. Line 18, indented once: Asterisk. blank spaces. calls Bank Account overloaded constructor. Line 19, indented once: Asterisk. blank spaces. at symbol, param balance starting balance. Line 20, indented once: Asterisk, forward slash. Line 21, indented once: public Checking Account, left parentheses, double balance, right parentheses. Line 22, indented once: left brace. Line 23, indented twice: super, left parentheses, balance, right parentheses, semicolon. 2 forward slashes. call Bank Account constructor. Line 24, indented twice: System dot out dot print l n, left parentheses, double quotes, In Checking Account, blank space, double quotes. Line 25, indented multiple times: plus, double quotes, overloaded constructor, double quotes, right parentheses, semicolon. Line 26, indented once: right brace. Line 27: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, new. Line 1: Forward slash. Asterisk. Checking Account Client, comma, Version 2. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Checking Account Client. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: 2 forward slashes. use default constructor. Line 10, indented twice: Checking Account c 1, equals, new Checking Account, left parentheses, right parentheses, semicolon. Line 11, indented twice: System dot out dot print l n, left parentheses, double quotes, New checking account, colon, double quotes, plus, c 1, plus, double quotes, backslash, n, double quotes, right parentheses, semicolon. Line 12: blank. Line 13, indented twice: 2 forward slashes. use overloaded constructor. Line 14, indented twice: Checking Account c 2, equals, new Checking Account, left parentheses, 100.00, right parentheses, semicolon. Line 15, indented twice: System dot out dot print l n, left parentheses, double quotes, New checking account, colon, blank space, double quotes, plus, c 2, right parentheses, semicolon. Line 16, indented once: right brace. Line 17: right brace.

Back to Example

The table has 4 columns labeled Superclass Members, Inherited by Subclass, question mark, Directly Accessible by Subclass, question mark, Directly Accessible by Client of Subclass Using a Subclass Reference, question mark. The row entry is as follows. Row 1. Superclass Members: constructors. Inherited by Subclass, question mark: no. Directly Accessible by Subclass, question mark: yes, using, super, left parentheses, a r g list, right parentheses, in a subclass constructor. Directly Accessible by Client of Subclass Using a Subclass Reference, question mark: no.

Back to Table

Program code. The code contains the following keywords: public, class, extends, final, double, private. Line 1: Forward slash. Asterisk. Checking Account class, comma, version 3. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank.

Line 5: public class Checking Account extends Bank Account. Line 6: left brace. Line 7, indented once: public final double DEFAULT underscore FEE, equals, 5.00, semicolon. Line 8, indented once: private double monthly Fee, semicolon. Line 9: blank. Line 10, indented once: Forward slash. 2 asterisks. default constructor. Line 11, indented once: Asterisk. Tab space. explicitly calls the Bank Account default constructor. Line 12, indented once: Asterisk. Tab space. sets monthly Fee to default value. Line 13, indented once: Asterisk. Forward slash. Line 14, indented once: public Checking Account, left parentheses, right parentheses. Line 15, indented once: left brace. Line 16, indented twice: super, left parentheses, right parentheses, semicolon. 2 forward slashes. optional. Line 17, indented twice: monthly Fee, equals, DEFAULT underscore FEE, semicolon. Line 18, indented once: right brace. Line 19: blank. Line 20, indented once: Forward slash. 2 asterisks. overloaded constructor. Line 21, indented once: Asterisk. calls Bank Account overloaded constructor. Line 22, indented once: Asterisk. at symbol, param, balance starting balance. Line 23, indented once: Asterisk. at symbol, param. monthly Fee starting monthly fee. Line 24, indented once: asterisk. Forward slash. Line 25, indented once: public Checking Account, left parentheses, double balance, comma. Line 26, indented multiple times: double monthly Fee, right parentheses. Line 27, indented once: left brace. Line 28, indented twice: super, left parentheses, balance, right parentheses, semicolon. 2 forward slashes, call Bank Account constructor. Line 29, indented twice: set Monthly Fee, left parentheses, monthly Fee, right parentheses, semicolon. Line 30, indented once: right brace. Line 31: blank. Line 32, indented once: Forward slash. 2 asterisks. apply Monthly Fee method. Line 33, indented once: asterisk, charges the monthly fee to the account. Line 34, indented once: asterisk, at symbol, return a reference to this object. Line 35, indented once: Asterisk, forward slash. Line 36, indented once: public Checking Account apply Monthly Fee, left parentheses, right parentheses. Line 37, indented once: left brace. Line 38, indented twice: withdraw, left parentheses, monthly Fee, right parentheses, semicolon. Line 39, indented twice: return this, semicolon. Line 40, indented once: right brace. Line 41: blank. Line 42, indented once: Forward slash, 2 asterisks. accessor method for monthly Fee. Line 43, indented once: asterisk, at symbol monthly Fee. Line 44, indented once: Asterisk. Forward slash. Line 45, indented once: public double get Monthly Fee, left parentheses, right parentheses. Line 46, indentd once: left brace. Line 47, indented twice: return monthly Fee, semicolon. Line 48, indented once: right brace. Line 49: blank. Line 50, indented once: Forward slash. 2 asterisks. mutator method for monthly Fee. Line 51, indented once: Asterisk. at symbol param monthly Fee new value for monthly Fee. Line 52, indented once: Asterisk. at symbol, return a reference to this obect. Line 53, indented once: Asterisk. Forward slash. Line 54, indented once: public Checking Account set Monthly Fee, left parentheses, double monthly Fee, right parentheses. Line 55, indented once: left brace. Line 56, indented twice: if, left parentheses, monthly Fee, greater than or equal to 0.0, right parentheses. Line 57, indented thrice: this dot monthly Fee, equals, monthly Fee, semicolon. Line 58: blank. Line 59, indented twice: return this, semicolon. Line 60, indented once: right brace. Line 61: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, new. Line 1: Forward slash. Asterisk. Checking Account Client, comma, Version 3. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Checking Account Client. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: Checking Account c 3, equals, new, Checking Account, left parentheses, 100.00, comma, 7.50, right parentheses, semicolon. Line 10, indented twice: System dot out dot print l n, left parentheses, double quotes, New checking account, colon, backslash, n, double quotes. Line 11, indented multiple times: plus, c 3 dot to String, left parentheses, right parentheses. Line 12, indented multiple times: plus, double quotes, semicolon, monthly fee is, double quotes. Line 13, indented multiple times: plus, c 3 dot get Monthly Fee, left parentheses, right parentheses, semicolon. Line

14: blank. Line 15, indented twice: c 3 dot apply Monthly Fee, left parentheses, right parentheses, semicolon. 2 forward slashes. charge the fee to the account. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash n, After charging monthly fee, colon, backslash n, double quotes. Line 17, indented multiple times: plus, c 3 dot to String, left parentheses, right parentheses. Line 18, indented multiple times: plus, double quotes, semicolon, monthly fee is, double quotes. Line 19, indented multiple times: plus, c 3 dot get Monthly Fee, left parentheses, right parentheses, right parentheses, semicolon. Line 20, indented once: right brace. Line 21: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, extends, final, double, private, return, this, if. Line 1: Forward slash. Asterisk. Checking Account class, comma, version 4. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Checking Account extends Bank Account. Line 6: left brace. Line 7, indented once: public final double DEFAULT underscore FEE, equals, 5.00, semicolon. Line 8, indented once: private double monthly Fee, semicolon. Line 9: blank. Line 10, indented once: Forward slash. 2 asterisks. default constructor. Line 11, indented once: Asterisk, explicitly calls the Bank Account default constructor. Line 12, indented once: Asterisk. sets monthly Fee to default value. Line 13, indented once: Asterisk. Forward slash. Line 14, indented once: public Checking Account, left parentheses, right parentheses. Line 15, indented once: left brace. Line 16, indented twice: super, left parentheses, right parentheses, semicolon. 2 forward slashes, call Bank Account constructor. Line 17, indented twice: monthly Fee, equals, DEFAULT underscore FEE, semicolon. Line 18, indented once: right brace. Line 19: blank. Line 20, indented once: Forward slash. 2 asterisks. overloaded constructor. Line 21, indented once: Asterisk. calls Bank Account overloaded constructor. Line 22, indented once: Asterisk, at symbol, param balance starting balance. Line 23, indented once: Asterisk, at symbol, monthly Fee starting monthly fee. Line 24, indented once: Asterisk, Forward slash. Line 25, indented once: public Checking Account, left parentheses, double balance, comma. Line 26, indented multiple times: double monthly Fee, right parentheses. Line 27, indented once: left brace. Line 28, indented twice: super, left parentheses, balance, right parentheses, semicolon, 2 forward slashes, call Bank Account constructor. Line 29, indented twice: set Monthly Fee, left parentheses, monthly Fee, right parentheses, semicolon. Line 30, indented once: right brace. Line 31: blank. Line 32, indented twice: Forward slash, 2 asterisks, apply Monthly Fee method. Line 33, indented once: Asterisk, charges the monthly fee to the account. Line 34, indented once: Asterisk, at symbol, return a reference to this object. Line 35, indented once: Asterisk. Forward slash. Line 36, indented once: public Checking Account apply Monthly Fee, left parentheses, right parentheses. Line 37, indented once: left brace. Line 38, indented twice: withdraw, left parentheses, monthly Fee, right parentheses, semicolon. Line 39, indented twice: return this, semicolon. Line 40, indented once: right brace. Line 41: blank. Line 42, indented once: Forward slash, 2 asterisks, accessor method for monthly Fee. Line 43, indented once: Asterisk, at symbol, return monthly Fee. Line 44, indented one: Asterisk. Forward slash. Line 45, indented once: public double get Monthly Fee, left parentheses, right parentheses. Line 46, indented once: left brace. Line 47, indented twice: return monthly Fee, semicolon. Line 48, indented once: right brace. Line 49: blank. Line 50, indented once: Forward slash. 2 asterisks. mutator method for monthly Fee. Line 51, indented once: Asterisk, at symbol, param, monthly Fee new value for monthly Fee. Line 52, indented once: Asterisk, at symbol, return a reference to this object. Line 53, indented once: Asterisk, forward slash. Line 54, indented once: public Checking Account set Monthly Fee, left parentheses, double monthly Fee, right parentheses. Line 55, indented once: left brace. Line 56, indented twice: if, left parentheses, monthly Fee, greater than or equal to 0.0, right parentheses. Line 57, indented thrice: this dot monthly Fee, equals, monthly Fee, semicolon. Line 58: blank. Line 59, indented twice: return this, semicolon. Line 60, indented once: right brace. Line 61: blank. Line 62, indented once: Forward slash. Asterisk, to String method. Line 63, indented once: Asterisk, at symbol, return String containing formatted balance and monthly Fee. Line 64, indented once: Asterisk, invokes superclass to String to format balance. Line 65, indented once: Asterisk, forward slash. Line 66, indented once: At symbol, Override. Line 67, indented once: public String to String, left parentheses, right parentheses. Line 68, indented once: left brace. Line 69, indented twice: return super dot to String, left parentheses, right parentheses. Line 70, indented multiple times: plus, double quotes, semicolon, monthly fee is, double quotes, plus, MONEY dot format, left parentheses, monthly Fee, right parentheses, semicolon. Line 71, indented once: right brace. Line 72: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, new. Line 1: Forward slash. Asterisk. Checking Account Client, comma, Version 4. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Checking Account Client. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: Checking Account c 4, equals, new, Checking Account, left parentheses, 100.00, comma, 7.50, right parentheses, semicolon. Line 10, indented twice: System dot out dot print l n, left parentheses, double quotes, New checking account, colon, backslash n, double quotes, plus, c 4, right parentheses, semicolon. Line 11, indented once: right brace. Line 12: right brace.

Back to Example

The table has 4 columns labeled Superclass Members, Inherited by Subclass, question mark, Directly Accessible by Subclass, question mark, Directly Accessible by Client of Subclass using Subclass Reference, question mark. The row entries are as follows. Row 1. Superclass Members: public or protected inherited methods that have been overridden in the subclass. Inherited by Subclass, question mark: no. Directly Accessible by Subclass, question mark: yes, using, super dot method Name, left parentheses, a r g list, right parentheses. Directly Accessible by Client of Subclass using Subclass Reference, question mark: no.

Back to Table

The table has 4 columns labeled Method Names, Argument Lists, Return Types, and Directly Accessible by Subclass Client using a Subclass Object Reference, question mark. The row entries are as follows. Row 1. Overriding a public method. Method Names: identical. Argument Lists: identical. Return Types: identical. Directly Accessible by Subclass Client Using a Subclass Object Reference, question mark: only the subclass version can be called. Row 2. Overloading a public method. Method

Names: identical. Argument Lists: different in number or type of parameters. Return Types: identical. Directly Accessible by Subclass Client Using a Subclass Object Reference, question mark: all versions of the overloaded method can be called.

Back to Table

Program code. The code contains the following keywords: import, public, class, final, new, protected, double, return, if, this, String. Line 1: Forward slash, 2 asterisks. Bank Account class, comma, version 1. Line 2: Asterisk. Tab space. Anderson, comma, Franceschi. Line 3: Asterisk. Tab space. Represents a generic bank account. Line 4: Asterisk. Forward slash. Line 5: import java dot text dot Decimal Format, semicolon. Line 6: blank. Line 7: public class Bank Account. Line 8: left brace. Line 9, indented once: public final Decimal Format MONEY. Line 10, indented multiple times: equals, new Decimal Format, left parentheses, double quotes, dollar sign, hash, comma, 2 hashes, 0.00, double quotes, right parentheses, semicolon. Line 11, indented once: protected double balance, semicolon. Line 12: blank. Line 13, indented once: Forward slash. 2 asterisks. default constructor. Line 14, indented once: Asterisk. sets balance to 0.0. Line 15, indented once: Asterisk, Forward slash. Line 16, indented once: public Bank Account, left parentheses, right parentheses. Line 17, indented once: left brace. Line 18, indented twice: balance, equals, 0.0, semicolon. Line 19, indented once: right brace. Line 20: blank. Line 21, indented once: Forward slash. 2 asterisks. overloaded constructor. Line 22, indented once: Asterisk. Tab space. at symbol, param balance begining balance. Line 23, indented once: Asterisk. Forward slash. Line 24, indented once: public Bank Account, left parentheses, double balance, right parentheses. Line 25, indented once: left brace. Line 26, indented twice: deposit, left parentheses, balance, right parentheses, semicolon. Line 27, indented once: right brace. Line 28: blank. Line 29, indented once: Forward slash. 2 asterisks. accessor for balance. Line 30, indented once: Asterisk. Tab space. at symbol, return current account balance. Line 31, indented once: Asterisk. Forward slash. Line 32, indented once: public double get Balance, left parentheses, right parentheses. Line 33, indented once: left brace. Line 34, indented twice: return balance, semicolon. Line 35, indented once: right brace. Line 36: blank. Line 37, indented once: Forward slash. 2 asterisks. deposit amount to account. Line 38, indented once: Asterisk. Tab space. at symbol, param amount amount to deposit, semicolon. Line 39, indented once: Asterisk. Tab spaces. amount must be greater than or equal to 0.0. Line 40, indented once: Asterisk. Tab space. at symbol, return a reference to this object. Line 41, indented once: Asterisk. Forward slash. Line 42, indented once: public Bank Account deposit, left parentheses, double amount, right parentheses. Line 43, indented once: left brace. Line 44, indented twice: if, left parentheses, amount greater than or equal to 0.0, right parentheses. Line 45, indented twice: balance, plus, equals, amount, semicolon. Line 46: blank. Line 47, indented twice: return this, semicolon. Line 48, indented once: right brace. Line 49: blank. Line 50, indented once: Forward slash. 2 asterisks. withdraw amount from account. Line 41, indented once: Asterisk. Tab space. at symbol, param amount. Tab space. amount to withdraw, semicolon. Line 52, indented once: Asterisk. Tab spaces. amount must be greater than or equal to 0.0. Line 53, indented once: Asterisk. Tab spaces. amount must be less than or equal to balance. Line 54, indented once: Asterisk. Tab spaces. amount must be less than or equal to balance. Line 54, indented once: Asterisk. Tab spaces. at symbol, return a reference to this object. Line 55, indented once: Asterisk. Forward slash. Line 56, indented once: public Bank Account withdraw, left parentheses, double amount, right parentheses. Line 57, indented once: left brace. Line 58, indented twice: if, left parentheses, amount, greater than or equal to 0.0, 2 ampersands, amount less than or equal to balance, right parentheses. Line 59, indented thrice: balance, minus, equals, amount, semicolon. Line 60: blank. Line 61, indented twice: return this, semicolon. Line 62, indented once: right brace. Line 63: blank. Line 64, indented once: Forward slash. 2 asterisks. to String. Line 65, indented once: Asterisk, blank spaces. at symbol, return the balance formatted as money. Line 66, indented once: Asterisk. Forward slash. Line 67, indented once: At symbol. Override. Line 68, indented once: public String to String, left parentheses, right parentheses. Line 69, indented once: left brace. Line 70, indented twice: return, double quotes, balance is, double quotes, plus, MONEY dot format, left parentheses, balance, right parentheses, semicolon. Line 71, indented once: right brace. Line 72: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, extends, final, double, return, this, if, super. Line 1: Forward slash. Asterisk. Checking Account class, comma, version 5. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Checking Account extends Bank Account. Line 6: left brace. Line 7, indented once: public final double DEFAULT underscore FEE equals 5.00, semicolon. Line 8, indented once: private double monthly Fee, semicolon. Line 9: blank. Line 10, indented once: Forward slash. 2 asterisks. default constructor. Line 11, indented once: Asterisk. Tab space. explicitly calls the Bank Account default constructor. Line 12, indented once: Asterisk. Tab space. set monthly Fee to default value. Line 13, indented once: Asterisk. Forward slash. Line 14, indented once: public Checking Account, left parentheses, right parentheses. Line 15, indented once: left brace. Line 16, indented twice: super, left parentheses, right parentheses, semicolon. 2 forward slashes, call Bank Account constructor. Line 17, indented twice: monthly Fee, equals, DEFAULT underscore FEE, semicolon. Line 18, indented once: right brace. Line 19: blank. Line 20, indented once: Forward slash. 2 asterisks. overloaded constructor. Line 21, indented once: Asterisk. calls Bank Account overloaded constructor. Line 22, indented once: Asterisk. at symbol, param balance starting balance. Line 23, indented once: Asterisk, at symbol param monthly Fee starting monthly fee. Line 24, indented once: Asterisk. Forward slash. Line 25, indented once: public Checking Account, left parentheses, double balance, comma. Line 26, indented multiple times: double monthly Fee, right parentheses. Line 27, indented once: left brace. Line 28, indented twice: super, left parentheses, balance, right parentheses, semicolon. 2 forward slashes, call Bank Account constructor. Line 29, indented twice: set Monthly Fee, left parentheses, monthly Fee, right parentheses, semicolon. Line 30, indented once: right brace. Line 31: blank. Line 32, indented once: Forward slash. 2 asterisks. apply Monthly Fee method. Line 33, indented once: Asterisk. charges the monthly fee to the account. Line 34, indented once: Asterisk. at symbol, return a reference to this object. Line 35, indented once: Asterisk. Forward slash. Line 36, indented once: public Checking Account apply Monthly Fee, left parentheses, right parentheses. Line 37, indented once: left brace. Line 38, indented twice: balance, minus, equals, monthly Fee, semicolon. Line 39: blank. Line 40, indented twice: return this, semicolon. Line 41, indented once: right brace. Line 42: blank. Line 43, indented once: Forward slash. 2 asterisks. accessor method for monthly Fee. Line 44, indented once: Asterisk, at symbol, return monthly Fee.

Line 45, indented once: Asterisk. Forward slash. Line 46, indented once: public double get Monthly Fee, left parentheses, right parentheses. Line 47, indented once: left brace. Line 48, indented twice: return monthly Fee, semicolon. Line 49, indented once: right brace. Line 50: blank. Line 51, indented once: Forward slash. 2 asterisks. mutator method for monthly Fee. Line 52, indented once: Asterisk, at symbol, param monthly Fee new value for monthly Fee. Line 53, indented once: Asterisk, at symbol, return a reference to this object. Line 54, indented once: Asterisk, Forward slash. Line 55, indented once: public Checking Account set Monthly Fee, left parentheses, double monthly Fee, right parentheses. Line 56, indented once: left brace. Line 57, indented twice: if, left parentheses, monthly Fee greater than or equal to 0.0, right parentheses. Line 58, indented thrice: this dot monthly Fee, equals, monthly Fee, semicolon. Line 59: blank. Line 60, indented twice: return this, semicolon. Line 61, indented once: right brace. Line 62: blank. Line 63, indented once: Forward slash. Asterisk. to String method. Line 64, indented once: Asterisk. at symbol, return String containing formatted balance and monthly Fee. Line 65, indented once: Asterisk. invokes superclass to String to format balance. Line 66, indented once: Asterisk. Forward slash. Line 67, indented once: at symbol, Override. Line 68, indented once: public String to String, left parentheses, right parentheses. Line 69, indented once: left brace. Line 70, indented twice: return super dot to String, left parentheses, right parentheses. Line 71, indented multiple times: plus, double quotes, semicolon, monthly fee is, double quotes, plus, MONEY dot format, left parentheses, monthly Fee, right parentheses, semicolon. Line 72, indented once: right brace. Line 73: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, void, String, if. Line 1: Forward slash. Asterisk. Checking Account Client, comma, version 5. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Checking Account Client. Line 6: left brace. Line 7, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: Checking Account c 5, equals, new Checking Account, left parentheses, 100.00, comma, 7.50, right parentheses, semicolon. Line 10, indented twice: System dot out dot print l n, left parentheses, double quotes, new checking account, colon, backslash, n, double quotes, plus, c 5, right parentheses, semicolon. Line 11: blank. Line 12, indented twice: c 5 dot withdraw, left parentheses, 95, right parentheses, semicolon. Line 13, indented twice: System dot out dot print l n, left parentheses, double quotes, backslash, n, After withdrawing dollar sign, 95, colon, backslash, n, double quotes, plus, c 5, right parentheses, semicolon. Line 14: blank. Line 15, indented twice:

System dot out dot print l n, left parentheses, double quotes, backslash, n, Applying the monthly fee, colon, double quotes, right parentheses, semicolon. Line 16, indented twice: c 5 dot apply Monthly Fee, left parentheses, right parentheses, semicolon. Line 17, indented twice: if, left parentheses, c 5 dot get Balance, left parentheses, right parentheses, less than 0.0, right parentheses. Line 18, indented thrice: System dot out dot print l n, left parentheses, double quotes, Warning, colon, account is overdrawn. double quotes, right parentheses, semicolon. Line 19, indented twice: System dot out dot print l n, left parentheses, c 5, right parentheses, semicolon. Line 20, indented once: right brace. Line 21: right brace.

Back to Example

The table has 4 columns labeled Superclass Members, Inherited by Subclass, question mark, Directly Accessible by Subclass, question mark, Directly Accessible by Client Subclass, question mark. The row entries are as follows. Row 1. Superclass Members: public fields. Inherited by Subclass, question mark: yes. Directly Accessible by Subclass, question mark: yes, by using field name. Directly Accessible by Client Subclass, question mark: yes. Row 2. Superclass Members: public methods. Inherited by Subclass, question mark: yes. Directly Accessible by Subclass, question mark: yes, by calling method from other subclass methods. Directly Accessible by Client Subclass, question mark: yes, by calling method using subclass object reference. Row 3. Superclass Members: protected fields. Inherited by Subclass, question mark: yes. Directly Accessible by Subclass, question mark: yes, by using field name. Directly Accessible by Client Subclass, question mark: no, must use accessors and mutators. Row 4. Superclass Members: protected methods. Inherited by Subclass, question mark: yes. Directly Accessible by Subclass, question mark: yes, by calling method from subclass methods. Directly Accessible by Client Subclass, question mark: no. Row 5. Superclass Members: private fields. Inherited by Subclass, question mark: no. Directly Accessible by Subclass, question mark: no, must use accessors and mutators. Directly Accessible by Client Subclass, question mark: no, must use accessors and mutators. Row 6. Superclass Members: private methods. Inherited by Subclass, question mark: no. Directly Accessible by Subclass, question mark: no. Directly Accessible by Client Subclass, question mark: no. Row 7. Superclass Members: constructors. Inherited by Subclass, question mark: no. Directly Accessible by Subclass, question mark: yes, using, super, left parentheses, a r g list, right parentheses, in a subclass constructor. Directly Accessible by Client Subclass, question mark: no. Row 8. Superclass Members: public or protected inherited methods that have been overridden in the subclass. Inherited by Subclass, question mark: no. Directly Accessible by Subclass, question mark: yes, using, super dot method Name, left parentheses, a r g list, right parentheses. Directly Accessible by Client Subclass, question mark: no.

Back to Table

Program code. The code contains the following keywords: import, public, class, final, double, if, this, return. Line 1: Forward slash. Asterisk. Savings Account class. Line 2: Asterisk. Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot text dot Decimal Format, semicolon. Line 6: blank. Line 7: 2 forward slashes. 1. 5 asterisks. indicate that Savings Account inherits. Line 8: 2 forward slashes. Tab spaces. from Bank Account. Line 9: public class Savings Account. Line 10: left brace. Line 11, indented once: public final double DEFAULT underscore RATE, equals, decimal point, 0 3, semicolon. Line 12, indented once: 2 forward slashes. 2. 5 asterisks. define the private interest Rate instance variable. Line 13, indented once: 2 forward slashes, interest Rate, comma, a double, comma, represents an annual rate. Line 14: blank. Line 15: blank. Line 16, indented once: 2 forward slashes. 3. 5 asterisks. write the default constructor. Line 17, indented once: Forward slash. 2 asterisks. default constructor. Line 18, indented once: Asterisk. Tab space. explicitly calls the Bank Account default constructor. Line 19, indented once: Asterisk. set interest Rate to default value DEFAULT underscore RATE. Line 20, indented once: Asterisk. Print a message to System dot out indicating that the constructor is called. Line 22, indented once: Asterisk.

Forward slash. Line 23: blank. Line 24: blank. Line 25, indented once: 2 forward slashes. 4. 5 asterisks. write the overloaded constructor. Line 26, indented once: Forward slash. 2 asterisks. overloaded constructor. Line 27, indented once: Asterisk. explicitly calls Bank Account overloaded constructor. Line 28, indented once: Asterisk. call set Interest Rate method, comma, passing interest Rate. Line 29, indented once: Asterisk. print a message to System dot out indicating that. Line 30, indented once: Asterisk. constructor is called. Line 31, indented once: Asterisk, at symbol, param balance starting balance. Line 32, indented once: Asterisk, at symbol, param, interest Rate starting interest rate. Line 33, indented once: Asterisk. Forward slash. Line 34: blank. Line 35: blank. Line 36, indented once: 2 forward slashes. 5. 5 asterisks. Write this method, colon. Line 37, indented once: Forward slash. 2 asterisks. apply Interest method, comma, no parameters. Line 38, indented once: Asterisk. call the deposit method, comma, passing a moth's worth of interest. Line 39, indented once: Asterisk, remember that the interest Rate instance variable is annual rate. Line 40, indented once: Asterisk. Line 41, indented once: Asterisk, at symbol, return a reference to this object. Line 42, indented once: Asterisk. Forward slash. Line 43: blank. Line 44: blank. Line 45, indented once: Forward slash. 2 asterisks. accessor method for interest Rate. Line 46, indented once: Asterisk. at symbol, return, interest Rate. Line 47, indented once: Asterisk, Forward slash. Line 48, indented once: public double get Interest Rate, left parentheses, right parentheses. Line 49, indented once: left brace. Line 50, indented twice: return interest Rate, semicolon. Line 51, indented once: right brace. Line 52: blank. Line 53, indented once: Forward slash. 2 asterisks. mutator method for interest Rate. Line 54, indented once: Asterisk, at symbol, param, interest Rate new value for interest Rate. Line 55, indented once: Asterisk. interest Rate must be greater than or equal to 0.0. Line 56, indented once: Asterisk. if not, comma, do not change the value. Line 57, indented once: Asterisk. stores interest Rate as input value, forward slash, 100. Line 58, indented once: Asterisk. that is, comma, 3.5 is stored as decimal point, 0 3 5. Line 59, indented once: Asterisk, at symbol, return a reference to this object. Line 60, indented once: Asterisk. Forward slash. Line 61, indented once: public Savings Account set Interest Rate, left parentheses, double interest Rate, right parentheses. Line 62, indented once: left brace. Line 63, indented twice: if, left parentheses, interest Rate, greater than or equal to 0.0, right parentheses. Line 64, indented thrice: this dot interest Rate, equals, interest Rate, forward slash, 100, semicolon. Line 65: blank. Line 66, indented twice: return this, semicolon. Line 67, indented once: right brace. Line 68: blank. Line 69, indented once: 2 forward slashes, 6, 5 asterisks, write this method. Line 70, indented once: Forward slash, Asterisk, to String method. Line 71, indented once: Asterisk, at symbol, return a String containing formatted balance and interest Rate. Line 72, indented once: Asterisk, invokes superclass to String to format balance. Line 73, indented once: Asterisk, formats interest Rate as percent using a Decimal Format object. Line 74, indented once: Asterisk. To create a Decimal Format object for formatting percentages. Line 75, indented once: Asterisk, use this pattern in the constructor, colon, double quotes, 0.00 percent, double quotes. Line 76, indented once: Asterisk, Forward slash. Line 77: blank. Line 78: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, abstract, class, private, int, this, return, void. Line 1: Forward slash. 2 asterisks. Abstract Figure superclass for drawing shapes. Line 2: Asterisk. Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 5: import java f x dot scene dot paint dot Color, semicolon. Line 6: blank. Line 7: public abstract class Figure. Line 8: left brace. Line 9, indented once: private int x, semicolon. Line 10, indented once: private int y, semicolon. Line 11, indented once: private Color color, semicolon. Line 12: blank. Line 13, indented once: Forward slash. 2 asterisks, default constructor. Line 14, indented once: Asterisk. sets x and y to 0. Line 15, indented once: Asterisk, sets color to black. Line 16, indented once: Asterisk, Forward slash. Line 17, indented once: public Figure, left parentheses, right parentheses. Line 18, indented once: left brace. Line 19, indented twice: x equals 0, semicolon. Line 20, indented twice: y equals 0, semicolon. Line 21, indented twice: color, equals, Color dot BLACK, semicolon. Line 22, indented once: right brace. Line 23: blank. Line 24, indented once: Forward slash. 2 asterisks. overloaded constructor. Line 25, indented once: Asterisk, at symbol, param x, starting x coordiante for figure. Line 26, indented once: Asterisk, at symbol, param y, starting y coordinate for figure. Line 27, indented once: Asterisk, at symbol, param, color, figure color. Line 28, indented once: Asterisk, Forward slash. Line 29, indented once: public Figure, left parentheses, int x, comma, int y, comma, Color color, right parentheses. Line 30, indented once: left brace. Line 31, indented twice: this dot x, equals, x, semicolon. Line 32, indented twice: this dot y equals y, semicolon. Line 33, indented twice: this dot color, equals, color, semicolon. Line 34, indented once: right brace. Line 35: blank. Line 36, indented once: Forward slash. 2 asterisks. accessor method for color. Line 37, indented once: Asterisk, at symbol, return current figure color. Line 38, indented once: Asterisk, forward slash. Line 39, indented once: public Color get Color, left parentheses, right parentheses. Line 40, indented once: left brace. Line 41: blank. Line 42, indented twice: return color, semicolon. Line 43, indented once: right brace. Line 44: blank. Line 45, indented once: Forward slash. 2 asterisks, mutator method for color. Line 46, indented once: Asterisk, at symbol, param color new color for figure. Line 47, indented once: Asterisk, at symbol, return, a reference to this object. Line 48, indented once: Asterisk. Forward slash. Line 49, indented once: public Figure set Color, left parentheses, Color color, right parentheses. Line 50, indented once: left brace. Line 51, indented twice: this dot color, equals, color, semicolon. Line 52: blank. Line 53, indented twice: return this, semicolon. Line 54, indented once: right brace. Line 55: blank. Line 56, indented once: Forward slash. 2 asterisks, accessor method for x. Line 57, indented once: Asterisk, at symbol, return current x value. Line 58, indented once: Asterisk, Forward slash. Line 59, indented once: public int get X, left parentheses, right parentheses. Line 60, indented once: left brace. Line 61, indented twice: return x, semicolon. Line 62, indented once: right brace. Line 63: blank. Line 64, indented once: Forward slash. 2 asterisks, mutator method for x. Line 65, indented once: Asterisk. at symbol, param, x, new value for x. Line 66, indented once: Asterisk, at symbol, return a reference to this object. Line 67, indented once: Asterisk. Forward slash. Line 68, indented once: public Figure, set X, left parentheses, int x, right parentheses. Line 69, indented once: left brace. Line 70, indented twice: this dot x, equals, x, semicolon. Line 71: blank. Line 72, indented twice: return this, semicolon. Line 73, indented once: right brace. Line 74: blank. Line 75, indented once: Forward slash, 2 asterisks, accessor method for y. Line 76, indented once: Asterisk, at symbol, return current y value. Line 77, indented once: Asterisk, Forward slash. Line 78, indented once: public int get Y, left parentheses, right parentheses. Line 79, indented once: left brace. Line 80, indented twice: return y, semicolon. Line 81, indented once: right brace. Line 82: blank. Line 83, indented once: Forward slash, 2 asterisks, mutator method for y. Line 84, indented once: Asterisk, at symbol, param y new y value. Line 85, indented once: Asterisk, at symbol, return a reference to this object. Line 86, indented once: Asterisk, Forward slash. Line 87, indented once: public Figure set Y, left parentheses, int y, right parentheses. Line 88, indented once: left brace. Line 89, indented twice: this dot y, equals, y, semicolon. Line 90: blank. Line 91, indented twice: return this, semicolon. Line 92, indented once: right brace. Line 93: blank. Line 94, indented once: Forward slash. 2 asterisks. abstract draw method. Line 95, indented once: Asterisk, at symbol, param, g c Graphics Context for drawing figure. Line 96, indented once: Asterisk, Forward slash. Line 97, indented once: public abstract void draw, left parentheses, Graphics Context g c, right parentheses, semicolon. Line 98: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, private, final, int, if, else, this, return, void. Line 1: Forward slash, asterisk, Circle class. Line 2: Asterisk, inherits from abstract Figure class. Line 3: Asterisk, Anderson, comma, Franceschi. Line 4: Asterisk, Forward slash. Line 5: blank. Line 6: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 7: import java f x dot scene dot paint dot Color, semicolon. Line 8: blank. Line 9: public class Circle extends Figure. Line 10: left brace. Line 11, indented once: private final int MIN underscore RADIUS, equals, 15, semicolon. Line 12, indented once: private int radius, semicolon. Line 13: blank. Line 14, indented once: Forward slash, 2 asterisks, default constructor. Line 15, indented once: Asterisk, calls default constructor of Figure class. Line 16, indented once: Asterisk, sets radius to minimum value. Line 17, indented once: Asterisk. Forward slash. Line 18, indented once: public Circle, left parentheses, right parentheses. Line 19, indented once: left brace. Line 20, indented twice: super, left parentheses, right parentheses, semicolon. Line 21, indented twice: radius, equals, MIN underscore RADIUS, semicolon. Line 22, indented once: right brace. Line 23: blank. Line 24, indented once: Forward slash, 2 asterisks. overloaded constructor. Line 25, indented once: Asterisk, sends x, comma, y, comma, color parameters to Figure constructor. Line 26, indented once: Asterisk, sends radius to set Radius method. Line 27, indented once: Asterisk, at symbol, param, x, starting x coordinate. Line 28, indented once: Asterisk, at symbol, y, starting y coordinate. Line 29, indented once: Asterisk, at symbol, param, color, color for circle. Line 30, indented once: Asterisk, at symbol, param, radius, radius of circle. Line 31, indented once: Asterisk, forward slash. Line 32, indented once: public Circle, left parentheses, int x, comma, int y, comma, Color color, comma. Line 33, indented multiple times: int radius, right parentheses. Line 34, indented once: left brace. Line 35, indented twice: super, left parentheses, x, comma, y, comma, color, right parentheses, semicolon. Line 36, indented twice: set Radius, left parentheses, radius, right parentheses, semicolon. Line 37, indented once: right brace. Line 38: blank. Line 39, indented once: Forward slash, 2 asterisks, mutator method for radius. Line 40, indented once: Asterisk, at symbol, param, radius new value for radius. Line 41, indented once: Asterisk, at symbol, return a reference to this object. Line 42, indented once: Asterisk, forward slash. Line 43, indented once: public Circle set Radius, left parentheses, int radius, right parentheses. Line 44, indented once: left brace. Line 45, indented twice: if, left parentheses, radius greater than MIN underscore RADIUS, right parentheses. Line 46, indented thrice: this dot radius, equals, radius, semicolon. Line 47, indented twice: else. Line 48, indented thrice: this dot radius, equals, MIN underscore RADIUS, semicolon. Line 49: blank. Line 50, indented twice: return this, semicolon. Line 51, indented once: right brace. Line 52: blank. Line 53, indented once: Forward slash, 2 asterisks, accessor method for radius. Line 54, indented once: Asterisk, at symbol, return radius. Line 55, indented once: Asterisk, Forward slash. Line 56, indented once: public int get Radius, left parentheses, right parentheses. Line 57, indented once: left brace. Line 58, indented twice: return radius, semicolon. Line 59, indented once: right brace. Line 60: blank. Line 61, indented once: Forward slash, 2 asterisks, draw method. Line 62, indented once: Asterisk, sets color and draws a circle. Line 63, indented once: Asterisk, at symbol, param, g c, Graphics Context for drawing the circle. Line 64, indented once: asterisk, forward slash. Line 65, indented once: at symbol, Override. Line 66, indented once: public void draw, left parentheses, Graphics Context g c, right parentheses. Line 67, indented once: left brace. Line 68, indented twice: g c dot set Fill, left parentheses, get Color, left parentheses, right parentheses, right parentheses, semicolon. Line 69, indented twice: g c dot fill Oval, left parentheses, get X, left parentheses, right parentheses, comma, get Y, left parentheses, right parentheses, comma. Line 70, indented multiple times: radius asterisk 2, comma, radius asterisk 2, right parentheses, semicolon. Line 71, indented once: right brace. Line 72: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, private, final, int, if, else, this, return, void. Line 1: Forward slash, asterisk. Square class. Line 2: Asterisk, inherits from abstract Figure class. Line 3: Asterisk, Anderson, Franceschi. Line 4: Asterisk, Forward slash. Line 5: blank. Line 6: import java f x dot scene dot canvas dot Graphics Context , semicolon. Line 7: import java f x dot scene dot paint dot Color, semicolon. Line 8: blank. Line 9: public class Square extends Figure. Line 10: Left brace. Line 11, indented once: private final int MIN underscore LENGTH equals 10 , semicolon. Line 12, indented once: private int length , semicolon. Line 13: blank. Line 14, indented once: Forward slash, asterisk.

Asterisk, default constructor. Line 15, indented once: Asterisk, calls default constructor of Figure class. Line 16, indented once: Asterisk, sets length to minimum value. Line 17, indented once: Asterisk, Forward slash. Line 18, indented once: public Square left parentheses, right parentheses. Line 19, indented once: Left brace. Line 20, indented twice: super left parentheses, right parentheses, semicolon. Line 21, indented twice: length equals MIN underscore LENGTH , semicolon. Line 22, indented once: Right brace. Line 23: blank. Line 24, indented once: Forward slash, asterisk. Asterisk, overloaded constructor. Line 25, indented once: Asterisk, sends x, y, and color parameters to Figure constructor. Line 26, indented once: Asterisk, sends length to set Length method. Line 27, indented once: Asterisk, at symbol param x starting x coordinate. Line 28, indented once: Asterisk, at symbol param y starting y coordinate. Line 29, indented once: Asterisk, at symbol param color color for square. Line 30, indented once: Asterisk, at symbol param length length of square. Line 31, indented once: Asterisk, Forward slash. Line 32, indented once: public Square left parentheses int x, int y, Color color. Line 33, indented multiple times: int length right parentheses. Line 34, indented once: Left brace. Line 35, indented twice: super left parentheses x, y, color right parentheses, semicolon. Line 36, indented twice: set Length left parentheses length right parentheses, semicolon. Line 37, indented once: Right brace. Line 38: blank. Line 39, indented once: Forward slash, asterisk. Asterisk, mutator method for length. Line

40, indented once: Asterisk, at symbol param length new value for length. Line 41, indented once: Asterisk, at symbol return a reference to this object. Line 42, indented once: Asterisk, Forward slash. Line 43, indented once: public Square set Length left parentheses int length right parentheses. Line 44: Left brace. Line 45, indented twice: if left parentheses length , right angled bracket, MIN underscore LENGTH right parentheses. Line 46, indented thrice: this dot length equals length , semicolon. Line 47, indented twice: else. Line 48, indented thrice: this dot length equals MIN underscore LENGTH , semicolon. Line 49: blank. Line 50, indented twice: return this, semicolon. Line 51, indented once: Right brace. Line 52: blank. Line 53, indented once: Forward slash, asterisk. Asterisk, accessor method for length. Line 54, indented once: Asterisk, at symbol return length. Line 55, indented once: Asterisk, Forward slash. Line 56, indented once: public int get Length left parentheses, right parentheses. Line 57: Left brace. Line 58, indented twice: return length , semicolon. Line 59: Right brace. Line 60: blank. Line 61, indented once: Forward slash, asterisk. Asterisk, draw method. Line 62, indented once: Asterisk, sets color and draws a square. Line 63, indented once: Asterisk, at symbol param g c Graphics Context for drawing the square. Line 64, indented once: Asterisk, Forward slash. Line 65, indented once: at symbol Override. Line 66, indented once: public void draw left parentheses Graphics Context g c right parentheses. Line 67, indented once: Left brace. Line 68, indented twice: g c dot set Fill left parentheses get Color left parentheses, right parentheses. right parentheses, semicolon. Line 69, indented once: g c dot fill Rect left parentheses get X left parentheses, right parentheses, get Y left parentheses, right parentheses. Line 70, indented multiple times: length, length, right parentheses, semicolon. Line 71, indented once: Right brace. Line 72: Right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, private, void, final, int, new, for, static. Line 1: Forward slash, asterisk. Figure Hierarchy Client. Line 2: Asterisk, Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: import java f x dot application dot Application, semicolon. Line 5: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 6: import java f x dot scene dot paint dot Color, semicolon. Line 7: import java f x dot stage dot Stage, semicolon. Line 8: blank. Line 9: import java dot util dot Array List, semicolon. Line 10: blank. Line 11: public class Traffic Light extends Application. Line 12: Left brace. Line 13, indented once: private Array List left angled bracket, Square, right angled bracket, squares List, semicolon. Line 14, indented once: private Array List left angled bracket, Circle, right angled bracket, circles List, semicolon. Line 15: blank. Line 16, indented once: at symbol Override. Line 17, indented once: public void start left parentheses Stage stage right parentheses. Line 18, indented once: Left brace. Line 19, indented twice:

Graphics Context g c equals J I Graphics Utility dot set Up Graphics left parentheses. Line 20, indented multiple times: stage, double quotes Traffic Light double quotes , 700, 400 right parentheses, semicolon. Line 21: blank. Line 22, indented twice: final int SQUARE underscore SIZE equals 80, CIRCLE underscore RADIUS equals 20, Line semicolon. Line 23, indented twice: final int SQUARE underscore X equals 300, CIRCLE underscore X equals SQUARE underscore X plus sign 20, semicolon. Line 24, indented twice: final int SQUARE underscore Y equals 100, CIRCLE underscore Y equals SQUARE underscore Y plus sign 20, semicolon. Line 25: blank. Line 26, indented twice: squares List equals new Array List left angled bracket, Square, right angled bracket, left parentheses, right parentheses, semicolon. Line 27, indented twice: squares List dot add left parentheses new Square left parentheses SQUARE underscore X, comma, SQUARE underscore Y, comma. Line 28, indented twice: Color dot BLACK, SQUARE underscore SIZE right parentheses right parentheses, semicolon. Line 29, indented twice: squares List dot add left parentheses new Square left parentheses SQUARE underscore X, comma, SQUARE underscore Y plus sign SQUARE underscore SIZE, comma. Line 30, indented multiple times: Color dot BLACK, SQUARE underscore SIZE right parentheses right parentheses, semicolon. Line 31: squares List dot add left parentheses new Square left parentheses SQUARE underscore X, comma, SQUARE underscore Y, plus sign, left parentheses SQUARE underscore SIZE

Asterisk, 2 right parentheses, comma. Line 32: Color dot BLACK, SQUARE underscore SIZE right parentheses right parentheses, semicolon. Line 33: blank. Line 34: circles List equals new Array List left angled bracket, Circle, right angled bracket, left parentheses, right parentheses, semicolon. Line 35, indented twice: circles List dot add left parentheses new Circle left parentheses CIRCLE underscore X, comma, CIRCLE underscore Y, comma. Line 36, indented multiple times: Color dot RED, CIRCLE underscore RADIUS right parentheses right parentheses, semicolon. Line 37, indented twice: circles List dot add left parentheses new Circle left parentheses CIRCLE underscore X, comma, CIRCLE underscore Y plus sign SQUARE underscore SIZE, comma. Line 38, indented multiple times: Color dot YELLOW, comma, CIRCLE underscore RADIUS right parentheses, right parentheses, semicolon. Line 39, indented twice: circles List dot add left parentheses new Circle left parentheses CIRCLE underscore X, comma, CIRCLE underscore Y plus sign left parentheses SQUARE underscore SIZE Asterisk, 2 right parentheses, comma. Line 40, indented multiple times: Color dot GREEN, comma, CIRCLE underscore RADIUS right parentheses right parentheses, semicolon. Line 41: blank. Line 42, indented twice: for left parentheses Square s, colon, squares List right parentheses. Line 43, indented thrice: s dot draw left parentheses g c, right parentheses, semicolon. Line 44: blank. Line 45, indented twice: for left parentheses Circle c, colon, circles List right parentheses. Line 46, indented thrice: c dot draw left parentheses g c right parentheses, semicolon. Line 47, indented once: Right brace. Line 48: blank. Line 49, indented once: public static void main left parentheses, String left square bracket, right square bracket, a r g s right parentheses. Line 50, indented once: Left brace. Line 51, indented twice: launch left parentheses, a r g s, right parentheses, semicolon. Line 52, indented once: Right brace. Line 53: Right brace.

Back to Example

The row entries are as follows. Row 1. abstract classes. Classes must be declared abstract if the class contains any abstract methods. Abstract classes can be extended. Abstract classes cannot be used to instantiate objects. Row 2. abstract methods within classes. abstract methods cannot be declared within a non-abstract class. An abstract method must consist of a method header followed by a semicolon. Abstract methods cannot be declared as private or static. A constructor cannot be declared abstract.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, private, void, final, int, new, static. Line 1: Forward slash, asterisk. Figure hierarchy Client. Line 2: Asterisk, Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: import java f x dot application dot Application, semicolon. Line 5: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 6: import java f x dot scene dot paint dot Color, semicolon. Line 7: import java f x dot stage dot Stage, semicolon. Line 8: blank. Line 9: import java dot util dot Array List, semicolon. Line 10: blank. Line 11: public class Traffic Light Polymorphism extends Application. Line 12: Left brace. Line 13, indented once: private Array List left angled bracket, Figure, right angled bracket, figures List, semicolon. Line 14: blank. Line 15, indented once: at symbol Override. Line 16, indented once: public void start left parentheses Stage stage right parentheses. Line 17, indented once: Left brace. Line 18, indented twice: Graphics Context g c equals J I Graphics Utility dot set Up Graphics left parentheses. Line 19, indented multiple times: stage, double quotes Traffic Light double quotes, 700, 400 right parentheses, semicolon. Line 20: blank. Line 21, indented twice: final int SQUARE underscore SIZE equals 80, CIRCLE underscore RADIUS equals 20 , Line semicolon. Line 22, indented twice: final int SQUARE underscore X equals 300, CIRCLE underscore X equals SQUARE underscore X plus sign 20, semicolon. Line 23, indented twice: final int SQUARE underscore Y equals 100, CIRCLE underscore Y equals SQUARE underscore Y plus sign 20, semicolon. Line 24: blank. Line 25, indented twice: figures List equals new Array List left angled bracket, Figure, right angled bracket, left parentheses, right parentheses, semicolon. Line 26: blank. Line 27, indented twice: figures List dot add left parentheses new Square left parentheses SQUARE underscore X, comma, SQUARE underscore Y, comma. Line 28, indented multiple times: Color dot BLACK, SQUARE underscore SIZE right parentheses, right parentheses, semicolon. Line 29, indented twice: figures List dot add left parentheses new Square left parentheses SQUARE underscore X, comma, SQUARE underscore Y plus sign SQUARE underscore SIZE, comma. Line 30, indented multiple times: Color dot BLACK, comma, SQUARE underscore SIZE right parentheses, right parentheses, semicolon. Line 31, indented twice: figures List dot add left parentheses new Square left parentheses SQUARE underscore X, comma, SQUARE underscore Y plus sign left parentheses SQUARE underscore SIZE Asterisk, 2 right parentheses, comma. Line 32, indented multiple times: Color dot BLACK, comma, SQUARE underscore SIZE right parentheses, right parentheses, semicolon. Line 33: blank. Line 34, indented twice: figures List dot add left parentheses new Circle left parentheses CIRCLE underscore X, comma, CIRCLE underscore Y, comma. Line 35, indented multiple times: Color dot RED, CIRCLE underscore RADIUS right parentheses, right parentheses, semicolon. Line 36, indented twice: figures List dot add left parentheses new Circle left parentheses CIRCLE underscore X, comma, CIRCLE underscore Y plus sign SQUARE underscore SIZE, comma. Line 37, indented multiple times: Color dot YELLOW, CIRCLE underscore RADIUS right parentheses, right parentheses, semicolon. Line 38, indented twice: figures List dot add left parentheses, new Circle left parentheses, CIRCLE underscore X, CIRCLE underscore Y plus sign, left parentheses SQUARE underscore SIZE Asterisk, 2, right parentheses. Line 39, indented multiple times: Color dot GREEN, CIRCLE underscore RADIUS right parentheses, right parentheses, semicolon. Line 40: blank. Line 41, indented twice: for left parentheses Figure f, colon, figures List, right parentheses. Line 42, indented thrice: f dot draw left parentheses g c right parentheses, semicolon. Line 43: blank. Line 44, indented once: Right brace. Line 45: blank. Line 46, indented once: public static void main left parentheses String left square bracket, right square bracket, a r g s, right parentheses. Line 47, indented once: Left brace. Line 48, indented twice: launch left parentheses a r g s right parentheses, semicolon. Line 49, indented once: Right brace. Line 50: Right brace.

Back to Example

Program code. The code contains the following keywords: import, public, abstract, class, private, String, int, this, return, void. Line 1: Forward slash, asterisk. Asterisk, Racer class. Line 2: Asterisk, Abstract class intended for racer hierarchy. Line 3: Asterisk, Anderson, Franceschi. Line 4: Asterisk, Forward slash. Line 5: blank. Line 6: import java f x dot scene dot canvas dot Graphics Context, semicolon. Line 7: blank. Line 8: public abstract class Racer. Line 9: Left brace. Line 10, indented once: private String I D, semicolon. 2 forward slashes racer I D. Line 11, indented once: private int x, semicolon. 2 forward slashes x position. Line 12, indented once: private int y, semicolon. 2 forward slashes y position. Line 13: blank. Line 14, indented once: Forward slash, asterisk. Asterisk, default constructor. Line 15, indented once: Asterisk, Sets I D to blank. Line 16, indented once: Asterisk, Forward slash. Line 17, indented once: public Racer left parentheses, right parentheses. Line 18, indented once: Left brace. Line 19, indented twice: I D equals double quotes double quotes, semicolon. Line 20, indented once: Right brace. Line 21: blank. Line 22, indented once: Forward slash, asterisk. Asterisk, constructor. Line 23, indented once: Asterisk, at symbol param I D racer I D. Line 24: Asterisk, at symbol param x, x position. Line 25, indented once: Asterisk, at symbol param y, y position. Line 26, indented once: Asterisk, Forward slash. Line 27, indented once: public Racer left parentheses String I D, int x, int y, right parentheses. Line 28, indented once: Left brace. Line 29, indented twice: this dot I D equals I D, semicolon. Line 30, indented twice: this dot x equals x, semicolon. Line 31, indented twice: this dot y equals y, semicolon. Line 32, indented once: Right brace. Line 33: blank. Line 34, indented once: Forward slash, asterisk. Asterisk, accessor for I D. Line 35, indented once: Asterisk, at symbol return I D. Line 36, indented once: Asterisk, Forward slash. Line 37, indented once: public String get I D, left parentheses, right parentheses. Line 38, indented once: Left brace. Line 39, indented twice: return I D, semicolon. Line 40, indented once: Right brace. Line 41: blank. Line 42, indented once: Forward slash, asterisk. Asterisk, accessor for x. Line 43, indented once: Asterisk, at symbol return current x value. Line 44, indented once: Asterisk, Forward slash. Line 45, indented once: public int get X, left parentheses, right parentheses. Line 46, indented once: Left brace. Line 47, indented twice: return x, semicolon. Line 48, indented once: Right brace. Line 49: blank. Line 50, indented once: Forward slash, asterisk. Asterisk, accessor for y. Line 51, indented once: Asterisk, at symbol return current y value. Line 52, indented once: Asterisk, Forward slash. Line 53, indented once: public int get Y left parentheses, right parentheses. Line 54, indented once: Left brace. Line 55, indented twice: return y, semicolon. Line 56, indented once: Right brace. Line 57: blank. Line 58, indented once: Forward slash, asterisk. Asterisk, mutator for x. Line 59, indented once: Asterisk, at symbol param x new value for x. Line 60, indented once: Asterisk, at symbol return a reference to this object. Line 61, indented once: Asterisk, Forward slash. Line 62, indented once: public Racer set X left parentheses int x right parentheses. Line 63, indented once: Left brace. Line 64, indented twice: this dot x equals x, semicolon. Line 65, indented twice: return this, semicolon. Line 66, indented once: Right brace. Line 67: blank. Line 68, indented once: Forward slash, asterisk. Asterisk, mutator for y. Line 69, indented once: Asterisk, at symbol param y new value for y. Line 70, indented once: Asterisk, at symbol return a reference to this object. Line 71, indented once: Asterisk, Forward slash. Line 72, indented once: public Racer set Y left parentheses int y right parentheses. Line 73, indented once: Left brace.

Line 74, indented twice: this dot y equals y, semicolon. Line 75, indented twice: return this, semicolon. Line 76, indented once: Right brace. Line 77: blank. Line 78, indented once: Forward slash, asterisk. Asterisk, abstract method for Racer's move. Line 79, indented once: Asterisk, Forward slash. Line 80, indented once: public abstract void move left parentheses, right parentheses, semicolon. Line 81: blank. Line 82, indented once: Forward slash, asterisk. Asterisk, abstract method for drawing Racer. Line 83, indented once: Asterisk, at symbol param g c Graphics Context. Line 84, indented once: Asterisk, Forward slash. Line 85: public abstract void draw left parentheses Graphics Context g c right parentheses, semicolon. Line 86: Right brace.

Back to Example

Program code. The code contains the following keywords: public, interface, void, char, default, void, String, for, int, if, else. Line 1: Forward slash, asterisk. String Handler interface. Line 2: Asterisk, Anderson, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public interface String Handler. Line 6: Left brace. Line 7, indented once: 2 forward slashes abstract methods to implement. Line 8, indented once: void process Letter left parentheses char c right parentheses, semicolon. Line 9, indented once: void process Digit left parentheses char c right parentheses, semicolon. Line 10, indented once: void process Other left parentheses char c right parentheses, semicolon. Line 11: blank. Line 12, indented once: Forward slash, asterisk. Asterisk, default parse method. Line 13, indented once: Asterisk, at symbol param s the String to parse. Line 14, indented once: Asterisk, Forward slash. Line 15, indented once: default void parse left parentheses String s right parentheses. Line 16, indented once: Left brace. Line 17, indented twice: for left parentheses int i equals 0 , semicolon. i left angled bracket, s dot length left parentheses, right parentheses, semicolon. i plus sign plus sign, right parentheses. Line 18, indented twice: Left brace. Line 19, indented thrice: char c equals s dot char At, left parentheses, I, right parentheses, semicolon. Line 20, indented thrice: if left parentheses Character dot is Digit left parentheses c right parentheses right parentheses. Line 21, indented 4 times: process Digit left parentheses c right parentheses, semicolon. Line 22, indented thrice: else if left parentheses Character dot is Letter left parentheses c right parentheses right parentheses. Line 23, indented 4 times: process Letter left parentheses c right parentheses, semicolon. Line 24, indented thrice: else. Line 25, indented 4 times: process Other left parentheses c right parentheses, semicolon. Line 26, indented twice: Right brace. Line 27, indented once: Right brace. Line 28: Right brace.

Back to Example

The title of the table reads: Useful Static Methods of the Character Class. The table has 2 columns labeled Return Value and Method name and argument list. The row entries are as follows. Row 1. Return value: boolean. Method name and argument list: is Digit, left parentheses, char c h, right parentheses, returns true if c h represents a digit from 0 to 9; false otherwise. Row 2. Return value: boolean. Method name and argument list: is Letter, char c h, right parentheses, returns true if c h represents a letter from lowercase a to lowercase z or uppercase a to uppercase z; false otherwise. Row 3. Return value: int. Method name and argument list: get Numeric Value, left parentheses, char c h, right parentheses, returns the integer value of the specified Unicode character c h.

Back to Table

Program code. The code contains the following keywords: public, class, implements, private, int, Boolean, false, true, void, char, String, if, else, return. Line 1: Forward slash, asterisk. Password Security Handler class. Line 2: Asterisk, Implements methods of the String Handler interface. Line 3: Asterisk, to parse a String containing a password. Line 4: Asterisk. Line 5: Asterisk, Anderson, Franceschi. Line 6: Asterisk, Forward slash. Line 7: blank. Line 8: public class Password Security Handler. Line 9, indented multiple times: implements String Handler. Line 10: Left brace. Line 11, indented once: private int length, semicolon. Line 12, indented once: private boolean digit, semicolon. Line 13, indented once: private boolean other Character, semicolon. Line 14: blank. Line 15, indented once: Forward slash, asterisk. Asterisk, default constructor. Line 16, indented once: Asterisk, sets length to 0, digit and other Character flags to false. Line 17, indented once: Asterisk, Forward slash. Line 18, indented once: public Password Security Handler left parentheses, right parentheses. Line 19, indented once: Left brace. Line 20, indented twice: length equals 0, semicolon. Line 21, indented twice: digit equals false, semicolon. Line 22, indented twice: other Character equals false, semicolon. Line 23, indented once: Right brace. Line 24: blank. Line 25, indented once: Forward slash, asterisk. Asterisk, process Letter method. Line 26, indented once: Asterisk, at symbol param c character to process. Line 27, indented once: Asterisk, adds 1 to length. Line 28, indented once: Asterisk, Forward slash. Line 29, indented once: at symbol Override. Line 30, indented once: public void process Letter, left parentheses, char c, right parentheses. Line 31, indented once: Left brace. Line 32, indented twice: length plus sign, plus sign, semicolon. Line 33, indented once: Right brace. Line 34: blank. Line 35, indented once: Forward slash, asterisk. Asterisk, process Digit method. Line 36, indented once: Asterisk, at symbol param c character to process. Line 37, indented once: Asterisk, adds 1 to length, sets digit flag to true. Line 38, indented once: Asterisk, Forward slash. Line 39, indented once: at symbol Override. Line 40, indented once: public void process Digit left parentheses char c right parentheses. Line 41, indented once: Left brace. Line 42, indented twice: length plus sign plus sign, semicolon. Line 43, indented twice: digit equals true, semicolon. Line 44, indented once: Right brace. Line 45:

blank. Line 46, indented once: Forward slash, asterisk. Asterisk, process Other method. Line 47, indented once: Asterisk, at symbol param c character to process. Line 48, indented once: Asterisk, adds 1 to length. Line 49, indented once: Asterisk, sets other Character flag to true. Line 50, indented once: Asterisk, Forward slash. Line 51, indented once: at symbol Override. Line 52, indented once: public void process Other left parentheses char c right parentheses. Line 53, indented once: Left brace. Line 54, indented twice: length plus sign plus sign, semicolon. Line 55, indented twice: other Character equals true, semicolon. Line 56, indented once: Right brace. Line 57: blank. Line 58, indented once: Forward slash, asterisk. Asterisk, security Level method. Line 59, indented once: Asterisk, at symbol return double quotes weak double quotes if password contains fewer than 6 characters. Line 60, indented once: Asterisk, double quotes strong double quotes if password has at least 8 characters, at least. Line 61, indented once: Asterisk, one digit, and at least one other character. Line 62, indented once: Asterisk, that is neither a letter nor a digit. Line 63, indented once: Asterisk, double quotes medium double quotes otherwise. Line 64, indented once: Asterisk, Forward slash. Line 65, indented once: public String security Level left parentheses, right parentheses. Line 66, indented once: Left brace. Line 67, indented twice: if left parentheses length left angled bracket, 6 right parentheses. Line 68, indented thrice: return double quotes weak double quotes, semicolon. Line 69, indented twice: else if left parentheses length, right angled bracket, equals 8, 2 ampersands, digit, 2 ampersands, other Character right parentheses. Line 70, indented thrice: return double quotes strong double quotes, semicolon. Line 71, indented twice: else. Line 72, indented thrice: return double quotes medium double quotes, semicolon. Line 73, indented once: Right brace. Line 74: Right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, new. Line 1: Forward slash. Asterisk. Password Security Handler Client. Line 2: Asterisk. Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Password Security Handler Client. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 12, indented twice: System dot out dot print l n, left parentheses, double quotes, A strong password has at least 8, backslash, n, double quotes. Line 13, indented multiple times: plus, double quotes, characters and contains at least one digit, backslash, n, double quotes. Line 14, indented multiple times: plus, double quotes, and one special character, period, double quotes, right parentheses, semicolon. Line 15, indented twice: System dot out dot print, left parentheses, double quotes, Enter a password, right angled bracket, double quotes, right parentheses, semicolon. Line 16, indented twice: String password, equals, scan dot next, left parentheses, right parentheses, semicolon. Line 17: blank. Line 18, indented twice: Password Security Handler p s h, equals, new Password Security Handler, left parentheses, right parentheses, semicolon. Line 19, indented once: p s h dot parse, left parentheses, password, right parentheses, semicolon. Line 20: blank. Line 21, indented twice: System dot out dot print l n, left parentheses, password, plus, double quotes, apostrophe, s, security is, double quotes. Line 22, indented multiple times: plus, p s h dot security Level, left parentheses, right parentheses, right parentheses, semicolon. Line 23, indented once: right brace. Line 24: right brace.

Back to Example

Program code. The code contains the following keywords: public, interface, boolean. Line 1: Forward slash. 2 asterisks. Validator interface. Line 2: Asterisk. Tab space. Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: at symbol, Functional Interface. Line 6: public interface validator. Line 7: left brace. Line 8, indented once: 2 forward slashes. abstract method to implement. Line 9, indented once: boolean is Valid, left parentheses, right parentheses, semicolon. Line 10: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, implements, private, Boolean, int, true, void, char, if, else, false, return. Line 1: Forward slash, asterisk, Hex String Handler class. Line 2: Asterisk, Implements the String Handler interface. Line 3: Asterisk, to parse a String that contains a hex number. Line 4: Asterisk, into its decimal equivalent. Line 5: Asterisk, Implements the Validator interface to determine. Line 6: Asterisk, validity of String. Line 7: Asterisk, Anderson, Franceschi. Line 8: Asterisk, Forward slash. Line 9: blank. Line 10: public class Hex String Handler. Line 11, indented multiple times: implements String Handler, Validator. Line 12: Left brace. Line 13, indented once: private boolean valid Hex, semicolon. Line 14, indented once: private int number, semicolon. Line 15: blank. Line 16, indented once: Forward slash, asterisk. Asterisk, default constructor. Line 17, indented once: Asterisk, initializes number to 0. Line 18, indented once: Asterisk, and valid Hex to true. Line 19, indented once: Asterisk, Forward slash. Line 20, indented once: public Hex String Handler left parentheses, right parentheses. Line 21, indented once: Left brace. Line 22, indented twice: valid Hex equals true, semicolon. Line 23, indented twice: number equals 0, semicolon. Line 24, indented once: Right brace. Line 25: blank. Line 26, indented once: Forward slash, asterisk. Asterisk, process Letter method. Line 27, indented once: Asterisk, at symbol param c the character to process. Line

28, indented once: Asterisk, if c is between 10 and 15 left parentheses hex A through F right parentheses. Line 29, indented once: Asterisk, uses its value to update the decimal value left parentheses number right Line parentheses. Line 30, indented once: Asterisk, otherwise, character is invalid letter. Line 31, indented once: Asterisk, Forward slash. Line 32, indented once: at symbol, Override. Line 33, indented once: public void process Letter left parentheses char c right parentheses. Line 34, indented once: Left brace. Line 35, indented twice: int n equals Character dot get Numeric Value left parentheses c right parentheses, semicolon. Line 36, indented twice: if left parentheses n, right angled bracket, equals 10, 2 ampersands, n left angled bracket, equals 15, right parentheses, 2 forward slashes, valid hex character, question mark. Line 37, indented thrice: number equals 16 Asterisk, number plus sign n, semicolon. 2 forward slashes update number. Line 38, indented twice: else 2 forward slashes invalid hex character. Line 39, indented thrice: valid Hex equals false, semicolon. Line 40, indented once: Right brace. Line 41: blank. Line 42, indented once: Forward slash, asterisk. Asterisk, process Digit method. Line 43, indented once: Asterisk, at symbol param c the character to process. Line 44, indented once: Asterisk, uses numeric value of c to update the decimal value left parentheses number right parentheses. Line 45, indented once: Asterisk, Forward slash. Line 46, indented once: at symbol Override. Line 47, indented once: public void process Digit left parentheses char c right parentheses. Line 48, indented once: Left brace. Line 49, indented twice: int n equals Character dot get Numeric Value left parentheses c right parentheses, semicolon. Line 50, indented twice: number equals 16 Asterisk, number plus sign n, semicolon. 2 forward slashes update number. Line 51, indented once: Right brace. Line 52: blank. Line 53, indented once: Forward slash, asterisk. Asterisk, process Other method. Line 54, indented once: Asterisk, at symbol param c the character to process. Line 55, indented once: Asterisk, character is not a valid hex digit. Line 56, indented once: Asterisk, Forward slash. Line 57, indented once: at symbol Override. Line 58, indented once: public void process Other left parentheses char c right parentheses. Line 59, indented once: Left brace. Line 60, indented twice: 2 forward slashes c is an invalid hex character. Line 61, indented twice: valid Hex equals false, semicolon. Line 62, indented once: Right brace. Line 63: blank. Line 64, indented once: Forward slash, asterisk. Asterisk, is Valid method. Line 65, indented once: Asterisk, at symbol return true if all characters. Line 66, indented once: Asterisk, in String are valid hex characters. Line 67, indented once: Asterisk, else returns false. Line 68, indented once: Asterisk, Forward slash. Line 69, indented once: public boolean is Valid left parentheses, right parentheses. Line 70, indented once: Left brace. Line 71, indented twice: return valid Hex, semicolon. Line 72, indented once: Right brace. Line 73: blank. Line 74, indented once: Forward slash, asterisk. Asterisk, get

Number method. Line 75, indented once: Asterisk, at symbol return if valid, returns the calculated decimal value. Line 76, indented once: Asterisk, else, returns negative 1. Line 77, indented once: Asterisk, Forward slash. Line 78, indented once: public int get Number left parentheses, right parentheses. Line 79, indented once: Left brace. Line 80, indented twice: if left parentheses is Valid left parentheses, right parentheses, right parentheses. Line 81, indented thrice: return number, semicolon. Line 82, indented twice: else. Line 83, indented thrice: return negative 1, semicolon. Line 84, indented once: Right brace. Line 85: Right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, new, if, else. Line 1: Forward slash. Asterisk. Hex String Handler Client class. Line 2: Asterisk. Tab space. Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Hex String Handler Client. Line 8, indented once: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 12, indented twice: System dot out dot print, left parentheses, double quotes, Enter a hexadecimal number, right angled bracket, double quotes, right parentheses, semicolon. Line 13, indented twice: String hex, equals, scan dot next, left parentheses, right parentheses, semicolon. Line 14: blank. Line 15, indented twice: Hex String Handler h s h, equals, new, Hex String Hander, left parentheses, right parentheses, semicolon. Line 16, indented twice: h s h dot parse, left parentheses, hex, right parentheses, semicolon. Line 17: blank. Line 18, indented twice: if, left parentheses, h s h dot is Valid, left parentheses, right parentheses, right parentheses. Line 19, indented thrice: System dot out dot print l n, left parentheses, hex, plus, double quotes, equals, double quotes, plus, h s h dot get Number, left parentheses, right parentheses, right parentheses, semicolon. Line 20, indented twice: else. Line 21, indented thrice: System dot out dot print l n, left parentheses, hex, plus, double quotes, is not a valid hex number, period, double quotes, right parentheses, semicolon. Line 22, indented once: right brace. Line 23: right brace.

Back to Example

Program code. The code contains the following keywords: public, interface, double. Line 1: Forward slash. 2 asterisks. Shipping Cost. Line 2: Asterisk. Tab space. Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: at symbol, Functional Interface. Line 6: public interface Shipping Cost. Line 7: left brace. Line 8, indented once: double calculate Shipping, left parentheses, double weight, right parentheses, semicolon. Line 9: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, implements, private, double, return. Line 1: Forward slash. 2 asterisks. Flat Rate Shipping. Line 2: Asterisk. Tab space. Cost is decimal point, 50, forward slash, pound. Line 3: Asterisk. Tab space. Anderson, comma, Franceschi. Line 4: Asterisk. Forward slash. Line 5: blank. Line 6: public class Flat Rate Shipping implements Shipping Cost. Line 7: left brace. Line 8, indented once: private final double RATE underscore PER underscore POUND, equals, decimal point, 50, semicolon. Line 9: blank. Line 10, indented once: at symbol, Override. Line 11, indented once: public double calculate Shipping, left parentheses, double weight, right parentheses. Line 12, indented once: left brace. Line 13, indented twice: return RATE underscore PER underscore POUND, asterisk, weight, semicolon. Line 14, indented once: right brace. Line 15: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, implements, private, final, double, if, else, return. Line 1: Forward slash. 2 asterisks. Standard Shipping. Line 2: Asterisk. Tab space. Cost is, dollar sign, 2.50, for the first 5 pounds, comma. Line 3: Asterisk. Tab spaces. and dollar sign, 1, for each additional pound. Line

4: Asterisk. Tab space. Anderson, comma, Franceschi. Line 5: Asterisk. Forward slash. Line 6: blank. Line 7: public class Standard Shipping implements Shipping Cost. Line 8: left brace. Line 9, indented once: private final double BASE underscore WEIGHT equals 5, semicolon. Line 10, indented once: private final double BASE underscore CHARGE, equals, 2.5, semicolon. Line 11, indented once: private final double EXTRA underscore POUND underscore RATE, equals, 1.00, semicolon. Line 12: blank. Line 13, indented once: at symbol, Override. Line 14, indented once: public double calculate Shipping, left parentheses, double weight, right parentheses. Line 15, indented once: left brace. Line 16, indented twice: if, left parentheses, weight greater than BASE underscore WEIGHT, right parentheses. Line 17, indented thrice: return BASE underscore CHARGE. Line 18, indented multiple times: plus, left parentheses, left parentheses, weight minus BASE underscore WEIGHT, right parentheses, asterisk, EXTRA underscore POUND underscore RATE, right parentheses, semicolon. Line 19, indented twice: else. Line 20, indented thrice: return BASE underscore CHARGE, semicolon. Line 21, indented once: right brace. Line 22: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, extends, private, final, double, return, super. Line 1: Forward slash. 2 asterisks. Express Shipping. Line 2: Asterisk. Tab space. Cost is dollar sign, 10, comma, in addition to standard shipping. Line 3: Asterisk. Tab space. Anderson, comma, Franceschi. Line 4: Asterisk. Forward slash. Line 5: blank. Line 6: public class Express Shipping extends Standard Shipping. Line 7: left brace. Line 8, indented once: private final double EXPRESS underscore SURCHARGE equals 10, semicolon. Line 9: blank. Line 10: at symbol, Override. Line 11, indented once: public double calculate Shipping, left parentheses, double weight, right parentheses. Line 12, indented once: left brace. Line 13, indented twice: return EXPRESS underscore SURCHARGE plus super dot calculate Shipping, left parentheses, weight, right parentheses, semicolon. Line 14, indented once: right brace. Line 15: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, new, double, int, switch, case, default, null, if, else. Line 1: Forward slash. 2 asterisk. Shipping Calculator. Line 2: Asterisk. Tab space. Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: import java dot util dot Scanner, semicolon. Line 5: import java dot text dot Number Format, semicolon. Line 6: blank. Line 7: public class Shipping Calculator. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 12, indented twice: Number Format money, equals, Number Format dot get Currency Instance, left parentheses, right parentheses, semicolon. Line 13: blank. Line 14, indented twice: System dot out dot print, left parentheses, double quotes, What is the weight of the item, question mark, blank space, double quotes, right parentheses, semicolon. Line 15, indented twice: double weight, equals, scan dot next Double, left parentheses, right parentheses, semicolon. Line 16: blank. Line 17, indented twice: System dot out dot print l n, left parentheses, double quotes, Our shipping options are, colon, blank space, double quotes. Line 18, indented multiple times: plus, double quotes, backslash, n, backslash t 1, Express shipping, colon, 1 hyphen 2 business days, double quotes. Line 19, indented multiple times: plus, double quotes, backslash, n, backslash, t 2, Standard shipping, colon, 3 hyphen 5 business days, double quotes. Line 20, indented multiple times: plus, double quotes, backslash, n, backslash, t 3, Flat Rate, colon, 6 hyphen 8 business days, double quotes, right parentheses, semicolon. Line 21, indented twice: System dot out dot print, left parentheses, double quotes, Select your shipping option, right angled bracket, double quotes, right parentheses, semicolon. Line 22, indented twice: int option, equals, scan dot next Int, left parentheses, right parentheses, semicolon. Line 23: blank. Line 24, indented twice: Shipping Cost shipping Cost, equals, switch, left parentheses, option, right parentheses. Line 25, indented twice: left brace. Line 26, indented thrice: case 1, hyphen, right angled bracket, new Express Shipping, left parentheses, right parentheses, semicolon. Line 27, indented thrice: case 2, hyphen, right angled bracket, new Standard Shipping, left parentheses, right parentheses, semicolon. Line 28, indented thrice: case 3, hyphen, right angled bracket, new Flat Rate Shipping, left parentheses, right parentheses, semicolon. Line 29, indented thrice: default, hyphen, right angled bracket, null, semicolon. Line 30, indented twice: right brace, semicolon. Line 31: blank. Line 32, indented twice: if, left parentheses, shipping Cost, exclamation mark, equals, null, right parentheses. Line 33, indented thrice: System dot out dot print l n, left parentheses, double quotes, Your shipping cost is, double quotes. Line 34, indented multiple times: plus, money dot format, left parentheses, shipping Cost dot calculate Shipping, left parentheses, weight, right parentheses, right parentheses, right parentheses, semicolon. Line 35, indented twice: else. Line 36, indented thrice: System dot out dot print l n, left parentheses, double quotes, No valid shipping option selected, period, double quotes, right parentheses, semicolon. Line 37, indented once: right brace. Line 38: right brace.

Back to Example

The table is titled, Methods of Exception Classes. The table has 2 columns labeled Return Value and Method Name and Argument List. The row entries are as follows.

Row 1. Return Value: String. Method Name And Exception List: get Message, left parentheses, right parentheses, returns a message indicating the cause of the exception. Row 2. Return Value: String. Method Name And Exception List: to String, left parentheses, right parentheses, returns a String containing the exception class name and a message indicating the cause of the exception. Row 3. Return Value: void. Method Name And Exception List: print Stack Trace, left parentheses, right parentheses, prints the line number of the code that caused the exception, along with the sequence of method calls leading up to the exception.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, String, try, while, int, catch. Line 1: Forward slash. Asterisk. Reading a Text File. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot i o dot File, semicolon. Line 6: import java dot i o dot File Not Found Exception, semicolon. Line 7: import java dot util dot Scanner, semicolon. Line 8: blank. Line 9: public class Echo File Data. Line 10: left brace. Line 11, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 12, indented once: left brace. Line 13, indented twice: try. Line 14, indented twice: left brace. Line 15, indented thrice: File input File, equals, new File, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, semicolon. Line 16, indented thrice: Scanner file, equals, new Scanner, left parentheses, input File, right parentheses, semicolon. Line 17: blank. Line 18, indented thrice: while, left parentheses, file dot has Next Int, left parentheses, right parentheses, right parentheses. Line 19, indented thrice: left brace. Line 20, indented 4 times: int number, equals, file dot next Int, left parentheses, right parentheses, semicolon. Line 21, indented 4 times: System dot out dot print l n, left parentheses, number, right parentheses, semicolon. Line 22, indented thrice: right brace. Line 23, indented twice: right brace. Line 24, indented twice: catch, left parentheses, File Not Found Exception f n f e, right parentheses. Line 25, indented twice: left brace. Line 26, indented thrice: System dot out dot print l n, left parentheses, double quotes, Message, colon, blank space, double quotes, plus, f n f e dot get Message, left parentheses, right parentheses, right parentheses, semicolon. Line 27, indented thrice: System dot out dot print l n, left parentheses, double quotes, backslash, n, to String, left parentheses, right parentheses, colon, blank space, double quotes, plus, f n f e, plus, double quotes, backslash, n, double quotes, right parentheses, semicolon. Line 28, indented thrice: f n f e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 29, indented twice: right brace. Line 30, indented once: right brace. Line 31: right brace.

Back to Example

The columns of the first table are labeled File Constructor and Exceptions Thrown. The row entry is as follows. Row 1. File Constructor: File, left parentheses, String path name, right parentheses, constructs a File object with the path name file name so that the file name is platform independent. Exceptions Thrown: Null Pointer Exception. The columns of the second table are labeled Scanner Constructor Methods and Exceptions Thrown. The row entry is as follows. Row 1. Scanner Constructor and Methods: Scanner, left parentheses, File file, right parentheses, constructs a Scanner object for reading from a file. Exceptions Thrown: File Not Found Exception. The columns of the third table are labeled Return Value, Method Name and Argument List, and Exceptions Thrown. The row entries are as follows. Row 1. Return Value: boolean. Method Name and Argument List: has Next, left parentheses, right parentheses, returns true if there is another token in the input stream. Return Value: boolean. Method Name and Argument List: has Next Line, left parentheses, right parentheses, returns true if there is another line or line separator in the input stream. Exceptions Thrown: Both methods throw an Illegal State Exception. Row 2. Return Value: boolean. Method Name and Argument List: has Next Int, left parentheses, right parentheses. Return Value: boolean. Method Name and Argument List: has Next Double, left parentheses, right parentheses. Return Value: boolean. Method Name and

Argument List: has Next Float, left parentheses, right parentheses, returns true if there is another token in the input stream of the specified data type. Exceptions Thrown: All these methods throw an Illegal State Exception. Row 3. Return Value: String. Method Name and Argument List: next, left parentheses, right parentheses, returns the next token in the input stream as a String. Return Value: String. Method Name and Argument List: next Line, left parentheses, right parentheses, returns the remainder of the line as a String. Positions the pointer to the next line. Exceptions Thrown: Both methods throw No Such Element Exception, Illegal State Exception. Row 4. Return Value: int. Method Name and Argument List: next Int, left parentheses, right parentheses. Return Value: double. Method Name and Argument List: next Double, left parentheses, right parentheses. Return Value: float. Method Name and Argument List: next Float, left parentheses, right parentheses, returns the next token in the input stream as the specified data type. Exceptions Thrown: All these methods throw Input Mismatch Exception, No Such Element Exception, Illegal State Exception. Return Value: void. Method Name and Argument List: close, left parentheses, right parentheses, releases the resources associated with an open input stream. Exceptions Thrown: None.

Back to Table

Program code. The code contains the following keywords:

import, public, class, static, void, String, try, new, while, if, else, int, catch. Line 1: Forward slash, 2 asterisks, Classic Movies, hyphen, Reading from a text file. Line 2: Asterisk. Anderson, comma, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: import java dot i o dot File, semicolon. Line 7: import java dot util dot No Such Element Exception, semicolon. Line 8: import java dot i o dot File Not Found Exception, semicolon. Line 9: blank. Line 10: public class Classic Movies. Line 11: left brace. Line 12, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 13, indented once: left brace. Line 14, indented twice: try, left parentheses, Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, movies dot t x t, double quotes, right parentheses, right parentheses, right parentheses. Line 15, indented twice: left brace. Line 16, indented thrice: while, left parentheses, file dot has Next, left parentheses, right parentheses, right parentheses, 2 forward slashes, test for the end of the file. Line 17, indented thrice: left brace. Line 18, indented 4 times: String movie Title, equals, file dot next Line, left parentheses, right parentheses, semicolon. Line 19: blank. Line 20, indented 4 times: if, left parentheses, exclamation mark, file dot has Next Int, left parentheses, right parentheses, right parentheses. Line 21, indented 4 times: left brace. Line 22, indented 5 times: System dot out dot print l n, left parentheses, double quotes, Invalid file format, double quotes, right parentheses, semicolon. Line 23, indented 5 times: String invalid Data, equals, file dot next Line, left parentheses, right parentheses, semicolon. 2 forward slashes, skip the line. Line 24, indented 4 times: right brace. Line 25, indented 4 times: else. Line 26, indented 4 times: left brace. Line 27, indented 5 times: int running Time, equals, file dot next Int, left parentheses, right parentheses, semicolon. Line 28, indented 5 times: String new Line, equals, file dot next Line, left parentheses, right parentheses, semicolon. 2 forward slashes, read new line character. Line 29, indented 5 times: System dot out dot print l n, left parentheses, movie Title, plus, double quotes, comma, blank space, double quotes. Line 30, indented multiple times: plus, running Time, plus, double quotes, minutes, double quotes, right parentheses, semicolon. Line 31, indented 4 times: right brace. Line 32, indented thrice: right brace. Line 33, indented twice: right brace. Line 34: blank. Line 35, indented twice: catch, left parentheses, File Not Found Exception f n f e, right parentheses. Line 36, indented twice: left brace. Line 37, indented thrice: System dot out dot print l n, left parentheses, double quotes, Unable to find movies dot t x t, comma, exiting, double quotes, right parentheses, semicolon. Line 38, indented twice: right brace. Line 39: blank. Line 40, indented twice: catch, left parentheses, No Such Element Exception n s e e, right parentheses. Line 41, indented twice: left brace. Line 42, indented thrice: System dot out dot print l n, left parentheses, double quotes, Attempt to read past the end of the file, double quotes, right parentheses, semicolon. Line 43, indented twice: right brace. Line 44, indented once: right brace. Line 45: right brace.

Back to Example

The table is titled, Input Classes. The table has 2 columns labeled Class and Description. The row entries are as follows. Row 1. Class: Scanner. Description: Class to read and parse characters in text files. Row 2. Class: Input Stream. Description: Abstract superclass representing an input stream of raw bytes. Row 3. Class: File Input Stream. Description: Input stream to read raw bytes of data from files. Row 4. Class: Object Input Stream. Description: Class to read, forward slash, recover objects from a file written using Object Output Stream.

Back to Table

The table is titled Output Classes. The table has 2 columns labeled Class and Description. The row entries are as follows. Row 1. Class: Writer. Description: Abstract superclass for writing characters to output streams. Row 2. Class: Output Stream. Description: Abstract superclass representing an output stream of raw bytes. Row 3. Class: File Writer. Description: Convenience class for writing characters to files. Row 4. Class: Print Writer. Description: Convenience class to output basic data types, Strings, and objects. Row 5. Class: File Output Stream.

Description: Output stream to write raw bytes of data to files. Row 6. Class: Object Output Stream. Description: Class to write objects to a file.

Back to Table

Program code. The code contains the following keywords: import, public, class, private, int, new, try, double, while, catch, finally, void, for, if. Line 1: Forward slash comma asterisk. GIF decoding class. Line 2: Anderson comma Franceschi. Line 3: Asterisk comma Forward slash. Line 4: blank. Line 5: import java.io. asterisk comma semicolon. Line 6: import java dot util dot asterisk comma semicolon. Line 7: import java f x dot scene dot canvas dot Graphics Context comma semicolon. Line 8: import java f x dot scene dot paint dot Color comma semicolon. Line 9: blank. Line 10: public class G I F Decoding. Line 11: Left brace. Line 12: private int width comma semicolon. 2 forward slashes the width of the G I F. Line 13: blank. Line 14: 2 forward slashes Array List instance variables colon. Line 15: 2 forward slashes color List stores Color references. Line 16: 2 forward slashes length List stores integers comma the length of each run. Line 17: private Array List left angled bracket Color right angled bracket color List comma semicolon. Line 18: private Array List left angled bracket Integer right angled bracket length List comma Line semicolon. Line 19: blank. Line 20: Forward slash comma asterisk. G I F Decoding constructor. Line 21: asterisk at symbol param file a Scanner comma already open for reading a G I F file. Line

22: asterisk initializes width with the first int in file. Line 23: asterisk fills the two Array Lists with the. Line 24: asterisk ints and colors stored in file. Line 25: Asterisk comma Forward slash. Line 26: public G I F Decoding left parentheses Scanner file right parentheses. Line 27: Left brace. Line 28: color List equals new Array List left angled bracket Color right angled bracket left parentheses right parentheses comma semicolon. Line 29: length List equals new Array List left angled bracket Integer right angled bracket left parentheses right parentheses comma semicolon. Line 30: blank. Line 31: try. Line 32: Left brace. Line 33: width equals file dot next Int left parentheses right parentheses comma semicolon. Line 34: int length comma semicolon. Line 35: double red comma green comma blue comma semicolon. Line 36: blank. Line 37: while left parentheses file dot has Next left parentheses right parentheses right parentheses. Line 38: Left brace. Line 39: length equals file dot next Int left parentheses right parentheses comma semicolon. Line 40: length List dot add left parentheses length right parentheses comma semicolon. Line 41: blank. Line 42: red equals file dot next Double left parentheses right parentheses comma semicolon. Line 43: green equals file dot next Double left parentheses right parentheses comma semicolon. Line 44: blue equals file dot next Double left parentheses right parentheses comma semicolon. Line 45: color List dot add left parentheses Color dot color left parentheses red comma green comma blue right parentheses right parentheses comma semicolon. Line 46: Right brace. Line 47: Right brace. Line 48: catch left parentheses Input Mismatch Exception i m e right parentheses. Line 49: Left brace. Line 50: System dot out dot print l n left parentheses double quotes Invalid file format comma semicolon. exiting double quotes right parentheses comma semicolon. Line 51: Right brace. Line 52: catch left parentheses exception e right parentheses. Line 53: Left brace. Line 54: System dot out dot print l n left parentheses e dot get Message left parentheses right parentheses right parentheses comma semicolon. Line 55: Right brace. Line 56: finally. Line 57: Left brace. Line 58: file dot close left parentheses right parentheses comma semicolon. Line 59: Right brace. Line 60: Right brace. Line 61: blank. Line 62: Forward slash comma asterisk. Line 63: asterisk draw G I F method. Line 64: asterisk at symbol param g c a Graphics Context reference comma the graphics context. Line 65: asterisk at symbol param x an int comma the drawing's upper left corner x coordinate. Line 66: asterisk at symbol param y an int comma the drawing's upper left corner y coordinate. Line 67: Asterisk comma Forward slash. Line 68: public void draw G I F left parentheses Graphics Context g c comma int x comma int y right parentheses. Line 69: Left brace. Line 70: 2 forward slashes draw the g i f. Line 71: Color current Color comma semicolon. Line 72: int index equals 0 comma semicolon. Line 73: int delta X equals 0 comma semicolon. Line 74: for left parentheses int length colon length List right parentheses. Line 75: Left brace. Line 76:

current Color equals color List dot get left parentheses index right parentheses comma semicolon. Line 77: g c dot set Stroke left parentheses current Color right parentheses comma semicolon. Line 78: blank. Line 79: 2 forward slashes draw one or more lines using length pixels with current Color. Line 80: while left parentheses length right angled bracket equals width minus delta X right parentheses. Line 81: Left brace. Line 82: g c dot stroke Line left parentheses x plus delta X comma y comma x plus width comma y right parentheses comma semicolon. Line 83: y plus plus comma semicolon. Line 84: length equals length minus left parentheses width minus delta X right parentheses comma semicolon. Line 85: delta X equals 0 comma semicolon. Line 86: Right brace. Line 87: blank. Line 88: if left parentheses length right angled bracket 0 right parentheses. Line 89: g c dot stroke Line left parentheses x plus delta X comma y comma x plus delta X plus length comma y right parentheses comma semicolon. Line 90: blank. Line 91: delta X equals delta X plus length comma semicolon. Line 92: index plus plus comma semicolon. Line 93: Right brace. Line 94: Right brace. Line 95: Right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, final, int, void, new, null, false, do, try, true, catch, while, static. Line 1: Forward slash comma asterisk. Drawing a G I F. Line 2: Anderson comma Franceschi. Line 3: Asterisk comma Forward slash. Line 4: blank. Line 5: import java.io. asterisk comma semicolon. Line 6: import java dot util dot Scanner comma semicolon. Line 7: import java f x dot application dot Application comma semicolon. Line 8: import java f x dot scene dot canvas dot Graphics Context comma semicolon. Line 9: import java f x dot scene dot paint dot Color comma semicolon. Line 10: import java f x dot stage dot Stage comma semicolon. Line 11: blank. Line 12: public class Draw A G I F extends Application. Line 13: Left brace. Line 14: final int WIDTH equals 500 comma semicolon. Line 15: final int HEIGHT equals 300 comma semicolon. Line 16: blank. Line 17: at symbol Override. Line 18: public void start left parentheses Stage stage right parentheses. Line 19: Left brace. Line 20: Scanner scan equals new Scanner left parentheses System.in right parentheses comma semicolon. Line 21: Scanner file equals null comma semicolon. Line 22: String filename equals double quotes double quotes comma semicolon. Line 23: boolean file Found equals false comma semicolon. Line 24: blank. Line 25: do. Line 26: Left brace. Line 27: try. Line 28: Left brace. Line 29: System dot out dot print left parentheses double quotes Enter the name of the file right angled bracket double quotes right parentheses comma semicolon. Line 30: filename equals scan dot next left parentheses right parentheses comma semicolon. Line 31: blank. Line 32: File input File equals new File left parentheses filename right parentheses comma semicolon. Line 33: file equals new Scanner left parentheses input File right parentheses comma semicolon. Line 34: file Found equals true comma semicolon. Line 35: Right brace. Line 36: catch left parentheses File Not Found Exception f n f right parentheses. Line 37: Left brace. Line 38: System dot out dot print l n left parentheses double quotes Unable to find double quotes plus filename right parentheses comma semicolon. Line 39: Right brace. Line 40: Right brace. while left parentheses exclamation mark file Found right parentheses comma semicolon. Line 41: blank. Line 42: 2 forward slashes set up window title and size. Line 43: GraphicsvContext gvc equals J I Graphics Utility dot set Up Graphics left parentheses. Line 44: stage comma double quotes Draw a flag double quotes comma WIDTH comma HEIGHT right parentheses comma semicolon. Line 45: g c dot set Fill left parentheses Color dot LIGHT SKY BLUE right parentheses comma semicolon. Line 46: g c dot fill Rect left parentheses 0 comma 0 comma WIDTH comma HEIGHT right parentheses comma semicolon. Line 47: blank. Line 48: G I F Decoding flag equals new G I F Decoding left parentheses file right parentheses comma semicolon. Line 49: flag dot draw G I F left parentheses g c comma 50 comma 50 right parentheses comma semicolon. Line 50: Right brace. Line 51: blank. Line 52: public static void main left parentheses String, left square bracket, right square bracket, a r g s right parentheses. Line 53: Left brace. Line 54: launch left parentheses a r g s right parentheses comma semicolon. Line 55: Right brace. Line 56: Right brace.

Back to Example

The table has 3 columns labeled Operation, If the File Exists, If the File Does Not Exist. The row entries are as follows. Row 1. Operation: write. If the File Exists: the current contents of the file are deleted, and writing starts at the beginning of the file. If the file Does Not Exist: the file is created and writing starts at the beginning of the file. Row 2. Operation: append. If the File Exists: data is added to the end of the file, keeping the existing contents. If the File Does Not Exist: the file is created and writing starts at the beginning of the file.

Back to Table

The table is titled, Classes, Constructors, and Methods for Writing to a Text File. The table has 3 columns labeled Class, Constructors, and Exceptions Thrown. The row entries are as follows. Row 1. Class: File Writer. Constructors: File Writer, left parentheses, String file name, comma, boolean mode, right parentheses, constructs a File Writer object from a String representing the name of a file; if mode is false, we will write to the file; if mode is true, we will append to the file. Exceptions thrown: I O Exception. Class: Print Writer. Constructors: Print Writer, left parentheses, Writer w r, right parentheses, constructs a Print Writer object from teh Writer object w r. Exceptions thrown: None. The method A P I's are as follows. Row 3. Class: Print Writer. Constructors: void print, left parentheses, int i, right parentheses, void print, left parentheses, double d, right parentheses, void print, left parentheses, char c, right parentheses, void print, left parentheses, boolean b, right parentheses, and so on, upto, void print l n, left parentheses, int i, right parentheses, void print l n, left parentheses, double d, right parentheses, and so on, writes the argument to a text file. Exceptions thrown: None. Row 4. Class: Print Writer. Constructors: void close, left parentheses, right parentheses, releases the resources associated with the Print Writer object. Exceptions thrown: None.

Back to Table

Program code. The code contains the following keywords: import, public, class, private, static, final, int, void, String, try, new, for, catch. Line 1: Forward slash. Asterisk. Writes a G I F encoding to a file. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot i o dot asterisk, semicolon. Line 6: blank. Line 7: public class write Flag To File. Line 8: left brace. Line 9, indented once: private static final int HEIGHT equals 10, semicolon. Line 10: blank. Line 11, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 12, indented once: left brace. Line 13, indented twice: try. Line 14, indented twice: left brace. Line 15, indented thrice: File Writer f w, equals, new File Writer, left parentheses, double quotes, Italian Flag dot t x t, double quotes, comma, false, right parentheses, semicolon. Line 16, indented thrice: Print Writer p w, equals, new Print Writer, left parentheses, f w, right parentheses, semicolon. Line 17, indented thrice: p w dot print l n, left parentheses, 30, right parentheses, semicolon. 2 forward slashes, width of G I F. Line 18, indented thrice: 2 forward slashes, write HEIGHT lines of color and pixels data. Line 19, indented thrice: for, left parentheses, int i equals 0, semicolon, i less than HEIGHT, semicolon, i plus, plus, right parentheses. Line 20, indented thrice: left brace. Line 21, indented 4 times: p w dot print, left parentheses, 10, right parentheses, semicolon. 2 forward slashes, 10 green pixels. Line 22, indented 4 times: p w dot print, left parentheses, single quote, blank space, single quote, right parentheses, semicolon. 2 forward slashes, white space character. Line 23, indented 4 times: p w dot print, left parentheses, 0.0, right parentheses, semicolon. 2 forward slashes, red color component. Line 24, indented 4 times: p w dot print l n, left parentheses, double quotes, blank space, 1.0, blank space, 0.0, double quotes, right parentheses, semicolon. 2 forward slashes, green and blue color components. Line 25: blank. Line 26, indented 4 times: p w dot print l n, left parentheses, double quotes, 10, blank space, 1.0, blank space, 1.0, blank space, 1.0, double quotes, right parentheses, semicolon. 2 forward slashes, 10 white pixels. Line 27, indented 4 times: p w dot print l n, left parentheses, double quotes, 10, blank space, 1.0, blank space, 0.0, blank space, 0.0, double quotes, right parentheses, semicolon. 2 forward slashes, 10 red pixels. Line 28, indented thrice: right brace. Line 29: blank. Line 30, indented thrice: p w dot close, left parentheses, right parentheses, semicolon. Line 31, indented twice: right brace. Line 32: blank. Line 33, indented twice: catch, left parentheses, I O Exception i o e, right parentheses. Line 34, indented twice: left brace. Line 35, indented thrice: System dot out dot print l n, left parentheses, i o e dot get Message, left parentheses, right parentheses, right parentheses, semicolon. Line 36, indented twice: right brace. Line 37, indented once: right brace. Line 38: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, final, int, static, void, String, int, new, Boolean, false, do, try, if, true, catch, while, for, int, catch. Line 1: Forward slash comma asterisk. Appends to a file. Line 2: Anderson comma Franceschi. Line 3: Asterisk comma Forward slash. Line 4: blank. Line 5: import java.io. asterisk comma semicolon. Line 6: import java dot util dot asterisk comma semicolon. Line 7: blank. Line 8: public class Append To Flag File. Line 9: Left brace. Line 10: static final int PIXELS equals 10 comma semicolon. Line 11: blank. Line 12: public static void main left parentheses String, left square bracket, right square bracket, a r g s right parentheses. Line 13: Left brace. Line 14: int lines equals 0 comma semicolon. Line 15: Scanner scan equals new Scanner left parentheses System.in right parentheses comma semicolon. Line 16: boolean good Input equals false comma semicolon. Line 17: blank. Line 18: do. Line 19: Left brace. Line 20: try. Line 21: Left brace. Line 22: 2 forward slashes prompt for input comma semicolon. expected value is an int. Line 23: System dot out dot print left parentheses double quotes Enter a number of lines left parentheses 20 to 50 right parentheses double quotes. Line 24: plus double quotes to append right angled bracket double quotes right parentheses comma semicolon. Line 25: lines equals scan dot next Int left parentheses right parentheses comma semicolon. Line 26: if left parentheses lines right angled bracket equals 20, 2 ampersands, lines left angled bracket equals 50 right parentheses. Line 27: good Input equals true comma semicolon. Line 28: Right brace. Line 29: blank. Line 30: catch left parentheses, Input Mismatch Exception i m e right parentheses. Line 31: Left brace. Line 32: 2 forward slashes consume invalid data left in input stream. Line 33: String garbage equals scan dot next Line left parentheses right parentheses comma semicolon. Line 34: System dot out dot print left parentheses double quotes You did not enter an integer comma semicolon. double quotes. Line 35: plus double quotes please enter an integer right angled bracket double quotes right parentheses comma semicolon. Line 36: Right brace. Line 37: Right brace. while left parentheses exclamation mark good Input right parentheses comma semicolon. Line 38: blank. Line 39: try. Line 40: Left brace. Line 41: File Writer f w equals new File Writer left parentheses double quotes Italian

Flag dot t x t double quotes comma true right parentheses comma semicolon. Line 42: Print Writer p w equals new Print Writer left parentheses f w right parentheses comma semicolon. Line 43: 2 forward slashes append lines of color and pixels data for this flag. Line 44: for left parentheses int i equals 0 comma semicolon. i left angled bracket lines comma semicolon. i plus plus right parentheses. Line 45: Left brace. Line 46: p w dot print l n left parentheses PIXELS plus double quotes 0.0 1.0 0.0 double quotes right parentheses comma semicolon. 2 forward slashes PIXELS green pixels. Line 47: p w dot print l n left parentheses PIXELS plus double quotes 1.0 1.0 1.0 double quotes right parentheses comma semicolon. 2 forward slashes PIXELS white pixels. Line 48: p w dot print l n left parentheses PIXELS plus double quotes 1.0 0.0 0.0 double quotes right parentheses comma semicolon. 2 forward slashes PIXELS red pixels. Line 49: Right brace. Line 50: p w dot close left parentheses right parentheses comma semicolon. Line 51: Right brace. Line 52: blank. Line 53: catch left parentheses I O Exception i o e right parentheses. Line 54: Left brace. Line 55: System dot out dot print l n, left parentheses, i o e dot get Message left parentheses right parentheses right parentheses comma semicolon. Line 56: Right brace. Line 57: Right brace. Line 58: Right brace.

Back to Example

The constructor for the Scanner class is as follows. Row

1. Constructor: Scanner, left parentheses, String source, right parentheses, constructs a Scanner object that produces tokens from the specified String. Exceptions Thrown: None. The description of the Method A P I is as follows. Row 2. Return Value: Scanner. Method name and argument list: Use Delimiter, left parentheses, String pattern, right parentheses, sets this Scanner object's delimiters based on the specified pattern. Exceptions Thrown: None.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, String, try, new, catch. Line 1: Forward slash comma asterisk. Demonstrating how to parse a String with the Scanner class. Line 2: Anderson comma Franceschi. Line 3: Asterisk comma Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner comma semicolon. Line 6: import java dot util dot Input Mismatch Exception comma semicolon. Line 7: import java dot util dot No Such Element Exception comma semicolon. Line 8: blank. Line 9: public class Using Scanner To Parse A String. Line 10: Left brace. Line 11: public static void main left parentheses String, left square bracket, right square bracket, a r g s right parentheses. Line 12: Left brace. Line 13: String flight Record 1 equals double quotes A A 123 comma B W I comma S F O comma 235 comma 239.5 double quotes comma semicolon. Line 14: blank. Line 15: try left parentheses Scanner parse equals new Scanner left parentheses flight Record 1 right parentheses right parentheses. Line 16: Left brace. Line 17: 2 forward slashes set the delimiter to a comma. Line 18: parse dot use Delimiter left parentheses double quotes comma double quotes right parentheses comma semicolon. Line 19: blank. Line 20: System dot out dot print l n left parentheses parse dot next left parentheses right parentheses right parentheses comma semicolon. 2 forward slashes flight number. Line 21: System dot out dot print l n left parentheses parse dot next left parentheses right parentheses right parentheses comma semicolon. 2 forward slashes origin airport. Line 22: System dot out dot print l n left parentheses parse dot next left parentheses right parentheses right parentheses comma semicolon. 2 forward slashes destination airport. Line 23: System dot out dot print l n left parentheses parse dot next Int left parentheses right parentheses right parentheses comma semicolon. 2 forward slashes number of passengers. Line 24: System dot out dot print l n left parentheses parse dot next Double left parentheses right parentheses right parentheses comma semicolon. 2 forward slashes average ticket price. Line 25: Right brace. Line 26: blank. Line 27: catch left parentheses Input Mismatch Exception i m e right parentheses. Line 28: Left brace. Line 29: System dot out dot print l n left parentheses double quotes Error in data format double quotes right parentheses comma semicolon. Line 30: Right brace. Line 31: blank. Line 32: catch left parentheses No Such Element Exception n s e right parentheses. Line 33: Left brace. Line 34: System dot out dot print l n left parentheses double quotes No more tokens double quotes right parentheses comma semicolon. Line 35: Right brace. Line 36: blank. Line 37: catch left parentheses Illegal State Exception i s e right parentheses. Line 38: Left brace. Line 39: i s e dot print Stack Trace left parentheses right parentheses comma semicolon. Line 40: Right brace. Line 41: Right brace. Line 42: Right brace.

Back to Example

Program code. The code contains the following keywords: record, String, int, double. Line 1: Forward slash. Asterisk. The Flight Record class. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: record Flight, left parentheses, String flight Number, comma. Line 6, indented multiple times: String origin, comma, String destination, comma. Line 7, indented multiple times: int num Passengers, comma, double a v g Ticket Price, right parentheses, left brace, right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, static, void, String, new, try, while, int, double, catch, for. Line 1: Forward slash comma asterisk. Reading structured data from a text file. Line 2: Anderson comma Franceschi. Line 3: Asterisk comma

Forward slash. Line 4: blank. Line 5: import java dot i o dot File comma semicolon. Line 6: import java dot i o dot File Not Found Exception comma semicolon. Line 7: blank. Line 8: import java dot util dot Scanner comma semicolon. Line 9: import java dot util dot Input Mismatch Exception comma semicolon. Line 10: import java dot util dot Array List comma semicolon. Line 11: blank. Line 12: public class Read Flights. Line 13: Left brace. Line 14: public static void main left parentheses String, left square bracket, right square bracket, a r g s right parentheses. Line 15: Left brace. Line 16: 2 forward slashes instantiate Array List to hold Flight objects. Line 17: Array List left angled bracket Flight right angled bracket list Flights equals new Array List left angled bracket Flight right angled bracket left parentheses right parentheses comma semicolon. Line 18: blank. Line 19: try. Line 20: Left brace. Line 21: Scanner file equals new Scanner left parentheses new File left parentheses double quotes flights dot t x t double quotes right parentheses right parentheses comma semicolon. Line 22: blank. Line 23: while left parentheses file dot has Next left parentheses right parentheses right parentheses 2 forward slashes test for the end of the file. Line 24: Left brace. Line 25: 2 forward slashes read a line. Line 26: String string Read equals file dot next Line left parentheses right parentheses comma semicolon. Line 27: blank. Line 28: 2 forward slashes process the line read. Line 29: Scanner parse equals new Scanner left parentheses string Read right parentheses comma semicolon. Line 30: parse dot use Delimiter left parentheses double quotes comma double quotes right parentheses comma semicolon. Line 31: String flight Number equals parse dot next left parentheses right parentheses comma semicolon. Line 32: String origin equals parse dot next left parentheses right parentheses comma semicolon. Line 33: String destination equals parse dot next left parentheses right parentheses comma semicolon. Line 34: blank. Line 35: tryLine 36: Left brace. Line 37: int num Passengers equals parse dot next Int left parentheses right parentheses comma semicolon. Line 38: double a v g Ticket Price equals parse dot next Double left parentheses right parentheses comma semicolon. Line 39: blank. Line 40: Flight f r Temp equals new Flight left parentheses. Line 41: flight Number comma origin comma destination comma. Line 42: num Passengers comma a v g Ticket Price right parentheses comma semicolon. Line 43: blank. Line 44: 2 forward slashes add Flight o b j to list Flights. Line 45: list Flights dot add left parentheses f r Temp right parentheses comma semicolon. Line 46: Right brace. Line 47: blank. Line 48: catch left parentheses Input Mismatch Exception i m e right parentheses. Line 49: Left brace. Line 50: System dot out dot print l n left parentheses double quotes Error in flight record colon double quotes. Line 51: plus string Read plus double quotes comma semicolon. record ignored double quotes right parentheses comma semicolon. Line 52: Right brace. Line 53: Right brace. Line 54: blank. Line 55: 2 forward slashes release resources associated with flights dot t x t. Line 56: file dot close left parentheses right parentheses comma semicolon. Line 57: Right brace. Line 58: blank. Line 59: catch left parentheses File Not Found Exception f n f e right parentheses. Line 60: Left brace. Line 61: System dot out dot print l n left parentheses double quotes Unable to find flights.txt double quotes right parentheses comma semicolon. Line 62: Right brace. Line 63: blank. Line 64: catch left parentheses Exception i o e right parentheses. Line 65: Left brace. Line 66: i o e dot print Stack Trace left parentheses right parentheses comma semicolon. Line 67: Right brace. Line 68: blank. Line 69: 2 forward slashes print the Flights read. Line 70: for left parentheses Flight flight colon list Flights right parentheses. Line 71: System dot out dot print l n left parentheses flight right parentheses comma semicolon. Line 72: Right brace. Line 73: Right brace.

Back to Example

Line 1: public void balance Check Book, left parentheses, right parentheses. Line 2: Left brace. Line 3: 2 forward slashes. 5 asterisks Write the body of this method 5 asterisks. Line 4: 2 forward slashes. Line 5: 2 forward slashes. Using a while loop, read the file transactions dot t x t. Line 6: 2 forward slashes. The file transactions dot t x t contains money. Line 7: 2 forward slashes. transactions between you and your bank. Line 8: 2 forward slashes. Line 9: 2 forward slashes. You will need to call the method animate inside. Line 10: 2 forward slashes. the body of the loop reading the file contents. Line 11: 2 forward slashes. Line 12: 2 forward slashes. The animate method takes 3 arguments, colon. Line 13: 2 forward slashes. a String, representing the type of transaction. Line 14: 2 forward slashes. a double, representing the transaction money amount. Line 15: 2 forward slashes. a double, representing the new check book balance. Line 16: 2 forward slashes. So if these 3 variables are, colon. Line 17: 2 forward slashes. transaction Name, current Amount, and balance dot Line 18: 2 forward slashes. then the call to animate will be, colon. Line 19: 2 forward slashes. Line 20: 2 forward slashes. animate, left parentheses, transaction Name, comma, current Amount, comma, balance, right parentheses, semicolon. Line 21: 2 forward slashes. Line 22: 2 forward slashes. You should make that call in the body of your while. Line 23: 2 forward slashes. loop, after you have updated the check book balance. Line 24: 2 forward slashes. Line 25: 2 forward slashes. Line 26: 2 forward slashes. end of student code. Line 27: 2 forward slashes. Line 28: Right brace.

Back to Example

The table has 2 columns labeled Return Value and Method and Description. The row entries are as follows. Row 1. Return Value: long. Method and Description: count, left parentheses, right parentheses, returns the number of elements in this Stream. Row 2. Return Value: Stream. Method and Description: filter, left parentheses, Predicate, left angled bracket, question mark, super T, right angled bracket, predicate, right parentheses, returns a Stream made of the elements of this Stream that satisfy predicate. Row 3. Return Value: Optional. Method and Description: min, left parentheses, Comparator, left angled bracket, question mark, super T, right angled bracket, comparator, right parentheses, returns the minimum element of this Stream using the comparator. Row 4. Return Value: Optional. Method and Description: max, left parentheses, Comparator, left angled bracket, question mark, super T, right angled bracket, comparator, right parentheses, returns the maximum element of this Stream using the comparator. Row 5. Return Value: Double Stream. Method and Description: map To Double, left parentheses, To Double Function, left angled bracket, question mark, super T, right angled bracket, mapper, right parentheses, returns a Double Stream made up of the elements resulting from applying mapper to the Stream elements; map To Int and map To Long are similar methods.

Back to Table

The table has 2 columns labeled Interface and Abstract Method. The row entries are as follows. Row 1. Interface: Predicate. Abstract Method: boolean test, left parentheses, uppercase t, lowercase t, right parentheses, tests t and returns true or false. Row 2. Interface: Double Predicate. Abstract Method: boolean test, left parentheses, double d, right parentheses, tests d and returns true or false. Row 3. Interface: Comparator. Abstract Method: int compare, left parentheses, uppercase t, lowercase t 1, comma, uppercase t, lowercase t 2, right parentheses, compares lowercase t 1 and lowercase t 2 for order. Row 4. Interface: To Double Function. Abstract Method: double apply As Double, left parentheses, uppercase t, lowercase t, right parentheses, converts lowercase t to a double and returns that value.

Back to Table

Program code. The code contains the following keywords: import, public, class, implements, boolean, if, return, true, else, String, return. Line 1: Forward slash. Asterisk. Predicate testing whether a String starts with B. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: import java dot util dot function dot asterisk, semicolon. Line 5: blank. Line 6: public class Start With B Predicate, left angled bracket, uppercase t, right angled bracket, implements Predicate, left angled bracket, uppercase t, right angled bracket. Line 7: left brace. Line 8, indented once: public boolean test, left parentheses, uppercase t, lowercase t, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: if, left parentheses, exclamation mark, right parentheses, t instance of String, right parentheses, right parentheses. Line 11, indented thrice: return false, semicolon. Line 12, indented twice: else. Line 13, indented twice: left brace. Line 14, indented thrice: String s, equals, left parentheses, String, right parentheses, t, semicolon. Line

15, indented thrice: return s dot char At, left parentheses, 0, right parentheses, equals, equals, single quote, uppercase b, single quote, semicolon. Line 16, indented twice: right brace. Line 17, indented once: right brace. Line 18: right brace.

Back to Example

The table has 3 columns labeled Class, Method, and Exceptions Thrown. The row entries are as follows. Row 1. Class: Paths. Method: static Path get, left parentheses, String first, comma, String, dot, dot, dot, more, right parentheses, returns a Path constructed with the String parameters. Exceptions Thrown: Invalid Path Exception. Row 2. Class: Files. Method: static Stream, left angled bracket, String, right angled bracket, lines, left parentheses, Path p, right parentheses, reads all the lines from p and returns them as a Stream. Exceptions Thrown: I O Exception, comma, Security Exception. Row 3. Class: Array List. Method: Stream stream, left parentheses, right parentheses, returns a Stream consisting of the elements in this Array List. Exceptions Thrown: blank.

Back to Table

Line 1: Forward slash. Asterisk. Reading file data into a Stream and processing it. Line 2: asterisk Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot i o dot I O Exception, comma, semicolon. Line 6: import java dot n i o dot file dot asterisk, comma, semicolon. Line 7: import java dot text dot Number Format, comma, semicolon. Line 8: import java dot util dot stream dot Stream, comma, semicolon. Line 9: blank. Line 10: public class Filter And Process Stream. Line 11: Left brace. Line 12: public static void main left parentheses String left square bracket right square bracket a r g s right parentheses. Line 13: Left brace. Line 14: Number Format money equals Number Format dot get Currency Instance, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 15: String city File equals double quotes cities dot t x t double quotes, comma, semicolon. Line 16: String expense File equals double quotes expenses dot t x t double quotes, comma, semicolon. Line 17: try. Line 18: Left brace. Line 19: 2 forward slashes. build a Stream using cities dot t x t. Line 20: Stream left angled bracket String right angled bracket city Stream equals. Line 21: Files dot lines left parentheses Paths dot get left parentheses city File right parentheses right parentheses, comma, semicolon. Line 22: 2 forward slashes. select cities starting with a B and count them. Line 23: long count equals city Stream dot filter left parentheses city hyphen, right angled bracket, city dot char At left parentheses 0 right parentheses equals equals single quote B single quote right parentheses. Line 24: dot count, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 25: System dot out dot print l n left parentheses double quotes Number of cities starting with B: double quotes plus count right parentheses, comma, semicolon. Line 26: blank. Line 27: 2 forward slashes. build a Stream using expenses dot t x t. Line 28: Stream left angled bracket String right angled bracket number Stream equals. Line 29: Files dot lines left parentheses Paths dot get left parentheses expense File right parentheses right parentheses, comma, semicolon. Line 30: 2 forward slashes. select positive costs and add them up. Line 31: double sum equals. Line 32: number Stream dot map To Double left parentheses data hyphen, right angled bracket, Double dot parse Double left parentheses data right parentheses right parentheses. Line 33: dot filter left parentheses cost hyphen, right angled bracket, cost right angled bracket 0.0 right parentheses. Line 34: dot sum, comma, left parentheses, comma, right parentheses. , comma, semicolon. Line 35: System dot out dot print l n left parentheses double quotes backslash n Total expenses are double quotes plus money dot format left parentheses sum right parentheses right parentheses , comma, semicolon. Line 36: Right brace. Line 37: blank. Line 38: catch left parentheses Invalid Path Exception i p e right parentheses. Line 39: Left brace. Line 40: System dot out dot print l n left parentheses i p e dot get Message, comma, left parentheses, comma, right parentheses right parentheses, comma, semicolon. Line 41: Right brace. Line 42: blank. Line 43: catch left parentheses I O Exception i o e right parentheses. Line 44: Left brace. Line 45: System dot out dot print l n left parentheses double quotes Could not find file, colon, double quotes plus i o e dot get Message, comma, left parentheses, comma, right parentheses, right parentheses , comma, semicolon. Line 46: Right brace. Line 47: blank. Line 48: catch left parentheses Security Exception s e right parentheses. Line 49: Left brace. Line 50: System dot out dot print l n left parentheses s e dot get Message, comma, left parentheses, comma, right parentheses right parentheses, comma, semicolon. Line 51: Right brace. Line 52: Right brace. Line 53: Right brace.

Back to Example

The table has 2 columns labeled Return Value and Method and Description. The row entries are as follows. Row 1. Return Value: Double Stream. Method and Description: filter, left parentheses, Double Predicate predicate, right parentheses, returns a Double Stream made of the elements of this Double Stream that satisfy predicate. Line 2. Return Value: double. Method and Description: sum, left parentheses, right parentheses, returns the sum of the elements in this Double Stream.

Back to Table

The table has 3 columns labeled Class, Constructor or Method and Exceptions Thrown. The row entries are as follows. Row 1. Class: U R L. Constructor or Method: U R L, left parentheses, String u r l, right parentheses, constructs a U R L object. Exceptions Thrown: Malformed U R L Exception, will be thrown if u r l is not a properly formed U R L. Row 2. Class: U R L. Constructor or Method: Input Stream open Stream, left parentheses, right parentheses, creates and returns an Input Stream for this U R L. Exceptions Thrown: I O Exception. Row 3. Class: Scanner. Constructor or Method: Scanner, left parentheses, Input Stream i s, right parentheses, constructs a Scanner object that can read from the Input Stream i s. Exceptions Thrown: blank.

Back to Table

Program code. The code contains the following keywords: import, public, class, static, void, String, try, new, catch. Line 1: Forward slash. Asterisk. J S O N parsing. Line 2: Asterisk. Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import o r g dot j s o n dot asterisk, semicolon. Line 6: blank. Line 7: public class J s o n Parsing. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: try. Line 12, indented twice: left brace. Line 13, indented thrice: String j s o n 1, equals, double quotes, left square bracket, Paris, comma, New York, comma, Rio, right square bracket, double quotes, semicolon. Line 14, indented thrice: J S O N Array j s o n A r r equals new J S O N Array, left parentheses, j s o n 1, right parentheses, semicolon. Line 15, indented thrice: String city 1, equals, j s o n A r r dot get String, left parentheses, 1, right parentheses, semicolon. Line 16, indented thrice: System dot out dot print l n, left parentheses, double quotes, city 1 is, double quotes, plus, city 1, right parentheses, semicolon. Line 17: blank. Line 18, indented thrice: String j s o n 2, equals. Line 19, indented 4 times: double quotes, left brace, single quote, city, single quote, colon, single quote, Miami, single quote, comma, single quote, state, single quote, colon, single quote, F L, single quote, comma, single quote, country, single quote, colon, single quote, U S A, single quote, right brace, double quotes, semicolon. Line 20, indented thrice: J S O N Object j s o n O b j, equals, new J S O N Object, left parentheses, j s o n 2, right parentheses, semicolon. Line 21, indented thrice: String state, equals, j s o n O b j dot get String, left parentheses, double quotes, state, double quotes, right parentheses, semicolon. Line 21, indented thrice: String state, equals, j s o n O b j dot get String, left parentheses, double quotes, state, double quotes, right parentheses, semicolon. Line 22, indented thrice: System dot out dot print l n, left parentheses, double quotes, state is, double quotes, plus, state, right parentheses, semicolon. Line 23, indented twice: right brace. Line 24, indented twice: catch, left parentheses, J S O N Exception j e, right parentheses. Line 25, indented twice: left brace. Line 26, indented thrice: System dot out dot print l n, left parentheses, double quotes, exception, colon, blank space, double quotes, plus, j e dot to String, left parentheses, right parentheses, right parentheses, semicolon. Line 27, indented twice: right brace. Line 28, indented once: right

brace. Line 29: right brace.

Back to Example

The table has 3 columns labeled Class, Constructor or Method and Exceptions Thrown. The row entries are as follows. Row 1. Class: J S O N Object. Constructor or Method: J S O N Object, left parentheses, String j s o n, right parentheses, constructs a J S O N Object. Exceptions Thrown: J S O N Exception, will be thrown if j s o n cannot be parsed into a J S O N Object. Row 2. Class: blank. Constructor or Method: data Type get Data Type, left parentheses, String key, right parentheses, returns the value for that key. The data type can be a primitive data type or String. Exceptions Thrown: J S O N Exception, will be thrown if key is not found or the value for key is not compatible with the expected return type. Row 3. Class: J S O N Array. Constructor or Method: J S O N Array, left parentheses, String j s o n, right parentheses, constructs a J S O N Array. Exceptions Thrown: J S O N Exception, will be thrown if j s o n cannot be parsed into a J S O N Array. Row 4. Class: blank. Constructor or Method: int length, left parentheses, right parentheses, returns the number of values in this J S O N Array. Exceptions Thrown: blank. Row 5. Class: blank. Constructor or Method: J S O N Object get J S O N Object, left parentheses, int index, right parentheses, returns the J S O N Object at index. Exceptions Thrown: J S O N Exception, will be thrown if index is invalid or there is no J S O N Object at that index.

Back to Table

Program code. The code contains the following keywords: record, String, int. Line 1: Forward slash. Asterisk. The Movie Record class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: record Movie, left parentheses, String title, comma, int year, right parentheses, left brace, right brace.

Back to Example

Line 1: Forward slash. Asterisk. A class to analyze data from movies. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java.io. asterisk , comma, semicolon. Line 6: import java dot net dot asterisk, comma, semicolon. Line 7: import java dot util dot asterisk, comma, semicolon. Line 8: import o r g dot j s o n dot asterisk, comma, semicolon. Line 9: blank. Line 10: public class Movies. Line 11: Left brace. Line 12: public static void main, left parentheses, String a r g s, left square bracket, right square bracket, right parentheses. Line 13: Left brace. Line 14: String GITHUB equals, double quotes, h t t p s, colon, 2 forward slashes, raw dot github user content dot com, forward slash, double quotes, comma, semicolon. Line 15: String file Name equals GITHUB plus. Line 16: double quotes, p rust, forward slash, Wikipedia hyphen movie hyphen data, forward slash, master, forward slash, movies dot j s o n, double quotes, comma, semicolon. Line 17: blank. Line

18: try. Line 19: Left brace. Line 20: U R L u r l equals new U R L, left parentheses, file Name, right parentheses, comma, semicolon. Line 21: Input Stream i s equals u r l dot open Stream, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 22: Scanner file equals new Scanner left parentheses i s right parentheses, comma, semicolon. Line 23: String j s o n equals double quotes double quotes, comma, semicolon. Line 24: String s equals null, comma, semicolon. Line 25: while left parentheses file dot has Next, comma, left parentheses, comma, right parentheses, right parentheses. Line 26: Left brace. Line 27: s equals file dot next Line, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 28: j s o n plus equals s, comma, semicolon. Line 29: Right brace. Line 30: blank. Line 31: J S O N Array j s o n Array equals new J S O N Array left parentheses j s o n, right parentheses, comma, semicolon. Line 32: System dot out dot print l n, left parentheses, double quotes, There are double quotes plus j s o n Array dot length, comma, left parentheses, comma, right parentheses. Line 33: plus double quotes movies double quotes right parentheses, comma, semicolon. Line 34: blank. Line 35: 2 forward slashes, Build Array List of Movie Record objects. Line 36: Array List left angled bracket Movie right angled bracket movies equals new Array List left angled bracket Movie right angled bracket, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 37: blank. Line 38: for, left parentheses, int i equals 0, comma, semicolon, i less than j s o n Array dot length, comma, left parentheses, comma, right parentheses, comma, semicolon, i plus plus, right parentheses. Line 39: Left brace. Line 40: J S O N Object j s o n Movie equals j s o n Array dot get J S O N Object left parentheses i right parentheses, comma, semicolon. Line 41: String title equals j s o n Movie dot get String left parentheses double quotes title double quotes right parentheses, comma, semicolon. Line 42: int year equals j s o n Movie dot get Int, left parentheses, double quotes, year, double quotes, right parentheses, comma, semicolon. Line 43: if, left parentheses, title dot equals, left parentheses, double quotes, double quotes, right parentheses, right parentheses. Line 44: System dot out dot print l n, left parentheses, double quotes, Discarding, left parentheses, no title, right parentheses, double quotes, right parentheses, comma, semicolon. Line 45: else. Line 46: movies dot add, left parentheses, new Movie, left parentheses, title, comma, year, right parentheses, right parentheses, comma, semicolon. Line 47: Right brace. Line 48: blank. Line 49: 2 forward slashes. Ask user to enter a year. Line 50: Scanner scan equals new Scanner, left parentheses, System dot in, right parentheses, comma, semicolon. Line 51: System dot out dot print, left parentheses, double quotes, Please enter a year, right angled bracket, double quotes, right parentheses, comma, semicolon. Line 52: int year equals scan dot next Int, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 53: blank. Line 54: 2 forward slashes. Retrieve number of movies for that year. Line 55: long count equals movies dot stream, comma, left parentheses, comma, right parentheses. Line 56: dot filter left parentheses movie hyphen, right angled bracket, movie dot year, comma, left parentheses, comma, right parentheses, equals equals, year, right parentheses. Line 57: dot count, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 58: System dot out dot print l n, left parentheses, double quotes, There were, double quotes, plus count. Line 59: plus, double quotes, movies in, double quotes, plus, year, right parentheses, comma, semicolon. Line 60: Right brace. Line 61: catch, left parentheses Malformed U R L Exception m u r l e, right parentheses. Line 62: Left brace. Line 63: System dot out dot print l n, left parentheses, m u r l e dot get Message, comma, left parentheses, comma, right parentheses, right parentheses, comma, semicolon. Line 64: Right brace. Line 65: catch left parentheses I O Exception i o e, right parentheses. Line 66: Left brace. Line 67: System dot out dot print l n, left parentheses, i o e dot get Message, comma, left parentheses, comma, right parentheses, right parentheses, comma, semicolon. Line 68: Right brace. Line 69: catch, left parentheses, J S O N Exception j s o n e, right parentheses. Line 70: Left brace. Line 71: System dot out dot print l n, left parentheses, j s o n e dot get Message, comma, left parentheses, comma, right parentheses, right parentheses, comma, semicolon. Line 72: Right brace. Line 73: Right brace. Line 74: Right brace.

Back to Example

The table is titled Classes, Constructors, and Methods for Writing Objects to a File. The table has 3 columns labeled Class, Constructor, and Exceptions thrown. The row entries are as follows. Row 1. Class: File Output Stream. Constructor: File Output Stream, left parentheses, String file name, comma, boolean mode, right parentheses, constructs a File Output Stream object from a String representing the name of a file; if mode is false, we will write to the file; if mode is true, we will append to the file. Exceptions thrown: File Not Found Exception. Row 2. Class: Object Output Stream. Constructor: Object Output Stream, left parentheses, Output Stream out, right parentheses, creates an Object Output Stream that writes to the Output Stream out. Exceptions thrown: I O Exception. The description of the Method A P I is as follows. Class: Object Output Stream. Constructor: void write Object, left parentheses, Object o, right parentheses, writes the object argument to a file. That object must be an instance of a class that implements the Serializable interface. Otherwise, a run-time exception will be generated. Exceptions thrown: I O Exception, Not Serializable Exception, Invalid Class Exception.

Back to Table

Program code. The code contains the following keywords: import, record, String, int, double, implements. Line 1: Forward slash. Asterisk. The Flight 2 Record class. Line

2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot i o dot Serializable, semicolon. Line 6: blank. Line 7: record Flight 2, left parentheses, String flight Number, comma. Line 8, indented multiple times: String origin, comma, String destination, comma. Line 9, indented multiple times: int num Passengers, comma, double a v g Ticket Price, right parentheses. Line 10, indented multiple times: implements Serializable, left brace, right brace.

Back to Example

Line 1: Forward slash. Asterisk. Demonstrating how to write objects to a file. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot i o dot File Output Stream, comma, semicolon. Line 6: import java dot i o dot Object Output Stream, comma, semicolon. Line 7: import java dot i o dot File Not Found Exception, comma, semicolon. Line 8: import java dot i o dot I O Exception, comma, semicolon. Line 9: blank. Line 10: public class Writing Objects. Line 11: Left brace. Line 12: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 13: Left brace. Line 14: 2 forward slashes, instantiate the objects. Line 15: Flight 2 f r 1 equals new Flight 2 left parentheses, double quotes, A A 3 1, double quotes, comma, double quotes B W I, double quotes, comma, double quotes, S F O, double quotes, comma, 200, comma, 235.9, right parentheses, comma, semicolon. Line 16: Flight 2 f r 2, equals, new Flight 2, left parentheses, double quotes, C O 25, double quotes, comma, double quotes, L A X, double quotes, comma, double quotes, J F K, double quotes, comma, 225, comma, 419.9, right parentheses, comma, semicolon. Line 17: Flight 2, f r 3, equals, new Flight 2, left parentheses, double quotes, U S 57, double quotes, comma, double quotes, B W I, double quotes, comma, double quotes, D E N, double quotes, comma, 175, comma, 179.5, right parentheses, comma, semicolon. Line 18: blank. Line 19: try. Line 20: Left brace. Line 21: File Output Stream f o s equals new File Output Stream. Line 22: left parentheses, double quotes, objects, double quotes, comma, false, right parentheses, comma, semicolon. Line 23: 2 forward slashes, false means we will write to objects. Line 24: blank. Line 25: Object Output Stream, o o s, equals, new Object Output Stream, left parentheses, f o s, right parentheses, comma, semicolon. Line 26: blank. Line 27: 2 forward slashes, write the objects to the file. Line 28: o o s dot write Object, left parentheses, f r 1, right parentheses, comma, semicolon. Line 29: o o s dot write Object, left parentheses, f r 2, right parentheses, comma, semicolon. Line 30: o o s dot write Object, left parentheses, f r 3, right parentheses, comma, semicolon. Line 31: blank. Line 32: 2 forward slashes, release resources associated with the objects file. Line 33: o o s dot close, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 34: Right brace. Line 35: blank. Line 36: catch left parentheses File Not Found Exception f n f e, right parentheses. Line 37:

Left brace. Line 38: System dot out dot print l n, left parentheses, double quotes, Unable to write to objects, double quotes, right parentheses, comma, semicolon. Line 39: Right brace. Line 40: blank. Line 41: catch left parentheses I O Exception i o e, right parentheses. Line 42: Left brace. Line 43: i o e dot print Stack Trace, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 44: Right brace. Line 45: Right brace. Line 46: Right brace.

Back to Example

The table is titled Classes, Constructors, and Methods for Reading Objects from a File. The table has 3 columns labeled Class, Constructors, and Exceptions thrown. The row entries are as follows. Row 1. Class: File Input Stream. Constructors: File Input Stream, left parentheses, String file name, right parentheses, constucts a File Input Stream object from a String representing the name of a file. Exceptions thrown: File Not Found Exception. Row 2. Class: Object Input Stream. Constructors: Object Input Stream, left parentheses, Input Stream i n, right parentheses, constructs an Object Input Stream from the Input Stream i n. Exceptions thrown: I O Exception. The Method A P I is as follows. Row 3. Class: Object Input Stream. Constructors: Object read Object, left parentheses, right parentheses, reads the next object and returns it. The object must be an instance of a class that implements the Serializable interface. When the end of the file is reached, an E O F Exception is thrown.

Exceptions thrown: I O Exception, Class Not Found Exception, E O F Exception.

Back to Table

Line 1: Forward slash. Asterisk. Demonstrating how to read objects from a file. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot i o dot asterisk, comma, semicolon. Line 6: import java dot util dot Array List, comma, semicolon. Line 7: blank. Line 8: public class Reading Objects Into Stream. Line 9: Left brace. Line 10: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11: Left brace. Line 12: Array List, left angled bracket, Flight 2, right angled bracket, flights, equals, new Array List, left angled bracket, Flight 2, right angled bracket, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 13: try. Line 14: Left brace. Line 15: File Input Stream f i s, equals, new, File Input Stream, left parentheses, double quotes, objects, double quotes, right parentheses, comma, semicolon. Line 16: Object Input Stream o i s, equals, new Object Input Stream, left parentheses, f i s, right parentheses, comma, semicolon. Line 17: blank. Line 18: try. Line 19: Left brace. Line 20: while, left parentheses, true, right parentheses. Line 21: Left brace. Line 22: 2 forward slashes, read object, comma, type cast returned object to Flight 2. Line 23: Flight 2 temp equals left parentheses Flight 2, right parentheses, o i s dot read

Object, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 24: blank. Line 25: 2 forward slashes, add the Flight 2 object read to flights. Line 26: flights dot add, left parentheses, temp, right parentheses, comma, semicolon. Line 27: Right brace. Line 28: Right brace. 2 forward slashes. end inner try block. Line 29: blank. Line 30: catch, left parentheses, E O F Exception e o f e, right parentheses. Line 31: Left brace. Line 32: System dot out dot print l n, left parentheses, double quotes, End of the file reached, double quotes, right parentheses, comma, semicolon. Line 33: Right brace. Line 34: blank. Line 35: catch, left parentheses, Class Not Found Exception c n f e, right parentheses. Line 36: Left brace. Line 37: System dot out dot print l n, left parentheses, c n f e dot get Message, comma, left parentheses, comma, right parentheses, right parentheses, comma, semicolon. Line 38: Right brace. Line 39: blank. Line 40: finally. Line 41: Left brace. Line 42: System dot out dot print l n, left parentheses, double quotes, Closing file, double quotes, right parentheses, comma, semicolon. Line 43: o i s dot close, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 44: Right brace. Line 45: Right brace, 2 forward slashes, end outer try block. Line 46: blank. Line 47: catch, left parentheses, File Not Found Exception, f n f e, right parentheses. Line 48: Left brace. Line 49: System dot out dot print l n, left parentheses, double quotes, Unable to find objects, double quotes, right parentheses, comma, semicolon. Line 50: Right brace. Line 51: blank.

Line 52: catch, left parentheses, I O Exception, i o e, right parentheses. Line 53: Left brace. Line 54: i o e dot print Stack Trace, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 55: Right brace. Line 56: blank. Line 57: 2 forward slashes, calculate number of flights originating from B W I. Line 58: long count equals. Line 59: flights dot stream, comma, left parentheses, comma, right parentheses. Line 60: dot filter, left parentheses, flight hyphen, right angled bracket, flight dot origin, comma, left parentheses, comma, right parentheses, dot, equals left parentheses, double quotes B W I, double quotes, right parentheses, right parentheses. Line 61: dot count, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 62: System dot out dot print l n, left parentheses, double quotes, There are, double quotes, plus, count, plus, double quotes, flights from B W I, double quotes, right parentheses, comma, semicolon. Line 63: blank. Line 64: 2 forward slashes, calculate average number of passengers on all flights. Line 65: double a v g Num Passengers equals. Line 66: flights dot stream, comma, left parentheses, comma, right parentheses. Line 67: dot map To Int, left parentheses, Flight 2, colon, colon, num Passengers, right parentheses. Line 68: dot average, comma, left parentheses, comma, right parentheses. Line 69: dot get As Double, comma, left parentheses, comma, right parentheses, comma, semicolon. Line 70: System dot out dot print l n, left parentheses, double quotes, Average number of passengers, colon, double quotes. Line 71: plus a v g Num Passengers, right parentheses, comma, semicolon. Line 72: Right brace. Line 73: Right brace.

Back to Example

Line 1: public void balance Check Book, left parentheses, right parentheses.
Line 2: Left brace. Line 3: 2 forward slashes. Line 4: 2 forward slashes. 5 asterisks Student writes the body of this method 5 asterisks. Line 5: 2 forward slashes. Line 6: 2 forward slashes. Using a while loop, read the file transactions dot o b j. Line 7: 2 forward slashes. The file transactions dot o b j contains transaction objects. Line 8: 2 forward slashes. Line 9: 2 forward slashes. You will need to call the animate method inside. Line 10: 2 forward slashes. the body of the loop that reads the objects. Line 11: 2 forward slashes. Line 12: 2 forward slashes. The animate method takes 2 arguments, colon. Line 13: 2 forward slashes. a Transaction object, representing the transaction. Line 14: 2 forward slashes. a double, representing the new checkbook balance. Line 15: 2 forward slashes. So if these two variables are transaction and balance. Line 16: 2 forward slashes. then the call to animate will be, colon. Line 17: 2 forward slashes. Line 18: 2 forward slashes. animate left parentheses transaction, balance right parentheses, semicolon. Line 19: 2 forward slashes. Line 20: 2 forward slashes. You should make that call in the body of your while. Line 21: 2 forward slashes. loop, after you have updated the

checkbook balance. Line 22: 2 forward slashes. Line 23: 2 forward slashes. Line 24: 2 forward slashes. Line 25: 2 forward slashes. Line 26: 2 forward slashes. end of student code. Line 27: 2 forward slashes. Line 28: Right brace.

Back to Example

Program code. The code contains the following keywords: public, class, extends, String. Line 1: Forward slash. Asterisk. The Illegal Email Exception class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Illegal Email Exception extends Exception. Line 6: left brace. Line 7, indented once: public Illegal Email Exception, left parentheses, String message, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: super, left parentheses, message, right parentheses, semicolon. Line 10, indented once: right brace. Line 11: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, final, char, private, String, throws, if, else, this, new, int, return. Line 1: Forward slash. Asterisk. The Email Address class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Email Address. Line 6: left brace. Line 7, indented once: public static final char AT underscore SIGN, equals, single quote, at symbol, single quote, semicolon. Line 8, indented once: private String e mail, semicolon. Line 9: blank. Line 10, indented once: public Email Address, left parentheses, String e mail, right parentheses. Line 11, indented multiple times: throws Illegal Email Exception. Line 12, indented once: left brace. Line 13, indented twice: if, left parentheses, e mail dot index Of, left parentheses, AT underscore SIGN, right parentheses, exclamation mark, equals, negative 1, right parentheses. Line 14, indented thrice: this dot e mail equals e mail, semicolon. Line 15, indented twice: else. Line 16, indented thrice: throw new Illegal Email Exception. Line 17, indented multiple times: left parentheses, double quotes, Email address does not contain, double quotes, plus, AT underscore SIGN, right parentheses, semicolon. Line 18, indented once: right brace. Line 19: blank. Line 20, indented once: public String get Host, left parentheses, right parentheses. Line 21, indented once: left brace. Line 22, indented twice: int index, equals, e mail dot index Of, left parentheses, AT underscore SIGN, right parentheses, semicolon. Line 23, indented twice: return e mail dot sub string, left parentheses, index plus 1, comma, e mail dot length, left parentheses, right parentheses, right parentheses, semicolon. Line 24, indented once: right brace. Line 25: right brace.

Back to Example

Program code. The code contains the following keywords:

import, public, class, static, void, String, new, try, catch. Line 1: Forward slash. Asterisk. The E mail Checker class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Email Checker. Line 8: left brace. Line 9, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: Scanner scan, equals, new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 12, indented twice: System dot out dot print, left parentheses, double quotes, Enter your e mail address, right angled bracket, double quotes, right parentheses, semicolon. Line 13, indented twice: String my Email, equals, scan dot next, left parentheses, right parentheses, semicolon. Line 14, indented twice: try. Line 15, indented twice: left brace. Line 16, indented thrice: Email Address address, equals, new E mail Address, left parentheses, my E mail, right parentheses, semicolon. Line 17, indented thrice: System dot out dot print l n, left parentheses, double quotes, Your host is, double quotes, plus, address dot get Host, left parentheses, right parentheses, right parentheses, semicolon. Line 18, indented twice: right brace. Line 19, indented twice: catch, left parentheses, Illegal E mail Exception i e e, right parentheses. Line 20, indented twice: left brace. Line 21, indented thrice: System dot out dot print l n, left parentheses, i e e dot get Message, left parentheses, right parentheses, right parentheses, semicolon. Line 22, indented twice: right brace. Line 23, indented once: right brace. Line 24: right brace.

Back to Example

Line 1: Forward slash. Asterisk. Printing Hello World n times using recursion. Line 2: Anderson, Franceschi. Line 3 Asterisk. Forward slash. Line 4: blank. Line 5: public class Recursive Hello World. Line 6 Left brace. Line 7: public static void main, left parentheses, String , left square bracket, right square bracket, a r g s, right parentheses. Line 8: Left brace. Line 9: 2 forward slashes. print double quotes, Hello World, double quotes, 5 times using our recursive method. Line 10: print Hello World N Times, left parentheses, 5, right parentheses, semicolon. Line 11: Right brace. Line 12: blank. Line 13: 2 forward slashes. the recursive method. Line 14: public static void print Hello World N Times, left parentheses, int n, right parentheses. Line 15: Left brace. Line 16: if, left parentheses, n greater than 0, right parentheses. Line 17: Left brace. Line 18: 2 forward slashes. print double quotes, Hello World, double quotes once. Line 19: System dot out dot print l n, left parentheses, double quotes, Hello World, double quotes, right parentheses, semicolon. Line 20: blank. Line 21: 2 forward slashes. now print, double quotes, Hello World, double quotes, left parentheses, n minus 1, right parentheses, times. Line 22: print Hello World N Times, left parentheses, n minus 1, right parentheses, semicolon. Line 23: Right brace. Line 24: 2

forward slashes. if n is 0 or less, do nothing. Line 25: Right brace. Line 26: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. Computing the factorial of a number using recursion. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Recursive Factorial. Line 6: Left brace. Line 7: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8: Left brace. Line 9: 2 forward slashes. compute factorial of 5 and output it. Line 10: System dot out dot print l n, left parentheses, double quotes, Factorial, left parentheses, 5, right parentheses, is double quotes. Line 11: plus factorial, left parentheses, 5, right parentheses, right parentheses, semicolon. Line 12: Right brace. Line 13: blank. Line 14: Forward slash. Asterisk. Asterisk. recursive factorial method. Line 15: Asterisk. at symbol, param n a positive integer. Line 16: Asterisk. at symbol, return the factorial of n. Line 17: Asterisk. Forward slash. Line 18: public static int factorial, left parentheses, int n, right parentheses. Line 19: Left brace. Line 20: if, left parentheses, n less than or equal to 0, right parentheses, 2 forward slashes. base case. Line 21: return 1, semicolon. Line 22: else, 2 forward slashes. general case. Line 23: return, left parentheses, n Asterisk. factorial, left parentheses, n minus 1, right parentheses, right parentheses, semicolon. Line 24: Right brace. Line 25 Right brace.

Back to Example

Line 1: Forward slash. Asterisk. Tracing the calculation of the factorial. Line 2: of a number using recursion. Line 3: Anderson, Franceschi. Line 4: Asterisk. Forward slash. Line 5: import java dot util dot List, semicolon. Line 6: import java dot util dot stream dot Collectors, semicolon. Line 7: blank. Line 8: public class Recursive Factorial With Stack Trace. Line 9: Left brace. Line 10: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 11: Left brace. Line 12: 2 forward slashes. compute factorial of 5 and output it. Line 13: System dot out dot print l n, left parentheses, double quotes, back slash, n, Factorial, left parentheses, 5, right parentheses, is double quotes. Line 14: plus factorial, left parentheses, 5 , right parentheses, , right parentheses, , semicolon. Line 15: Right brace. Line 16: blank. Line 17: Forward slash. Asterisk. Asterisk. recursive factorial method. Line 18: Asterisk. at symbol, param n a positive integer. Line 19: Asterisk. at symbol, return the factorial of n. Line 20: Asterisk. Forward slash. Line 21: public static int factorial, left parentheses, int n, right parentheses. Line 22: Left brace. Line 23: Listless than Stack Walker dot Stack Frame, greater than, stack equals Stack Walker dot get Instance, left parentheses, right parentheses. Line 24: dot walk, left parentheses, s hyphen, right angled bracket, s dot skip, left parentheses, 1, right parentheses, dot collect, left parentheses, Collectors dot to List, left parentheses, right parentheses, right parentheses, right parentheses, semicolon. Line 25: blank. Line 26: System dot out dot print l n, left parentheses, double quotes, factorial, left parentheses, double quotes plus n plus double quotes, right parentheses, called double quotes, right parentheses, semicolon. Line 27: System dot out dot print l n, left parentheses, double quotes, back slash, t, Current Stack, double quotes, right parentheses, semicolon. Line 28: for, left parentheses, Stack Walker dot Stack Frame f, colon, stack, right parentheses. Line 29: System dot out dot print l n, left parentheses, double quotes, back slash, t, back slash, t, double quotes, plus f dot get Line Number, left parentheses, right parentheses, plus double quotes double quotes. Line 30: plus f dot get Method Name, left parentheses, right parentheses, right parentheses, semicolon. Line 31: blank. Line 32: if, left parentheses, n equals, equals, 0, right parentheses, 2 forward slashes, base case. Line 33: Left brace. Line 34: System dot out dot print l n, left parentheses, double quotes, back slash n, Base case detected back slash, n, double quotes, right parentheses, semicolon. Line 35: blank. Line 36: System dot out dot print l n, left parentheses, double quotes, factorial, left parentheses, double quotes plus n plus double quotes, right parentheses, returning 1, double quotes, right parentheses, semicolon. Line 37: System dot out dot print l n, left parentheses, double quotes, back slash, t, Current Stack, double quotes, right parentheses, semicolon. Line 38: for, left parentheses, Stack Walker dot Stack Frame f colon stack, right parentheses. Line 39:

System dot out dot print l n, left parentheses, double quotes, back slash, t, back slash, t, double quotes, plus f dot get Line Number, left parentheses, right parentheses, plus double quotes double quotes. Line 40: plus f dot get Method Name, left parentheses, right parentheses, right parentheses, semicolon. Line 41: blank. Line 42: return 1, semicolon. Line 43: Right brace. Line 44: else, 2 forward slashes. general case. Line 45: Left brace. Line 46: int factorial N Minus 1 equals factorial, left parentheses, n minus 1, right parentheses, semicolon. Line 47: blank. Line 48: System dot out dot print l n, left parentheses, double quotes, factorial, left parentheses, double quotes plus n plus double quotes, right parentheses, returning double quotes. Line 49: plus n plus double quotes Asterisk. double quotes plus factorial N Minus 1, right parentheses, semicolon. Line 50: System dot out dot print l n, left parentheses, double quotes, back slash, t, Current Stack, double quotes, right parentheses, semicolon. Line 51: for, left parentheses, Stack Walker dot Stack Frame f, colon, stack, right parentheses. Line 52: System dot out dot print l n, left parentheses, double quotes, back slash, t, back slash, t, double quotes, plus f dot get Line Number, left parentheses, right parentheses, plus double quotes, double quotes. Line 53: plus f dot get Method Name, left parentheses, right parentheses, right parentheses, semicolon. Line 54: blank. Line 55: return, left parentheses, n Asterisk. factorial N Minus 1, right parentheses, semicolon. Line 56: Right brace. Line 57: Right brace. Line 58: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. Computing the greatest common divisor using recursion. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Recursive G C D. Line 6: Left brace. Line 7: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8: Left brace. Line 9: 2 forward slashes. compute and output g c d of 474 and 162. Line 10: System dot out dot print l n, left parentheses, double quotes, The G C D of double quotes plus 474 plus double quotes and double quotes. Line 11: plus 162 plus double quotes is double quotes plus g c d, left parentheses, 474, 162, right parentheses, right parentheses, semicolon. Line 12: Right brace. Line 13: blank. Line 14: Forward slash. Asterisk. Asterisk. recursive g c d method. Line 15: Asterisk. at symbol, param dividend the first strictly positive integer. Line 16: Asterisk. at symbol, param divisor the second strictly positive integer. Line 17: Asterisk. at symbol, return the g c d of dividend and divisor. Line 18: Asterisk. Forward slash. Line 19: public static int g c d, left parentheses, int dividend, int divisor, right parentheses. Line 20: Left brace. Line 21: if, left parentheses, dividend percentage sign divisor equals equals 0, right parentheses, 2 forward slashes. base case. Line 22: return divisor, semicolon. Line 23: else 2 forward slashes. general case. Line 24: return, left parentheses, g c d, left parentheses, divisor, dividend percentage sign divisor, right parentheses, right parentheses, semicolon. Line 25: Right brace. Line 26: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. Computing the greatest common divisor using recursion. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: import java dot util dot List, semicolon. Line 5: import java dot util dot stream dot Collectors, semicolon. Line 6: blank. Line 7: public class Recursive G C D With Stack Trace. Line 8: Left brace. Line 9: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10: Left brace. Line 11: 2 forward slashes. compute g c d of 474 and 162 and output it. Line 12: System dot out dot print l n, left parentheses, double quotes, back slash, n, The GCD of double quotes plus 474 plus double quotes and double quotes. Line 13: plus 162 plus double quotes is double quotes plus g c d, left parentheses, 474, 162 , right parentheses, right parentheses, semicolon. Line 14: Right brace. Line 15: blank. Line 16: Forward slash. Asterisk. Asterisk. recursive g c d method with trace. Line 17: Asterisk. at symbol, param dividend the first strictly positive integer. Line 18: Asterisk. at symbol, param divisor the second strictly positive integer. Line 19: Asterisk. at symbol, return the g c d of dividend and divisor. Line 20: Asterisk. Forward slash. Line 21: public static int g c d, left parentheses, int dividend, int divisor, right parentheses. Line 22: Left brace. Line 23: Listless than Stack Walker dot Stack Frame greater than stack equals Stack Walker dot get Instance, left parentheses, right parentheses. Line 24: dot walk, left parentheses, s hyphen right angled bracket s dot skip, left parentheses, 1, right parentheses, dot collect, left parentheses, Collectors dot to List, left parentheses, right parentheses, right parentheses, right parentheses, semicolon. Line 25: blank. Line 26: blank. Line 27: System dot out dot print, left parentheses, double quotes, g c d, left parentheses, double quotes plus dividend plus double quotes, double quotes plus divisor plus double quotes, right parentheses, double quotes, right parentheses, semicolon. Line 28: System dot out dot print l n, left parentheses, double quotes double quotes plus dividend plus double quotes percentage sign double quotes plus divisor plus double quotes equals double quotes. Line 29: plus, left parentheses, dividend percentage sign divisor, right parentheses, right parentheses, semicolon. Line 30: System dot out dot print l n, left parentheses, double quotes, back slash, t, Current stack, double quotes, right parentheses, semicolon. Line 31: for, left parentheses, Stack Walker dot Stack Frame f, colon, stack, right parentheses. Line 32: System dot out dot print l n, left parentheses, double quotes, back slash, t, back slash, t, double quotes, plus f dot get Line Number, left parentheses, right parentheses. Line 33: plus double quotes double quotes plus f dot get Method Name, left parentheses, right parentheses, right parentheses, semicolon. Line 34: blank. Line 35: if, left parentheses, dividend percentage sign divisor equals equals 0, right parentheses, 2 forward slashes, base case. Line 36: Left brace. Line 37: System dot out dot print l n, left parentheses, double quotes, back slash, n, base case reached, returning double quotes. Line 38: plus divisor, right parentheses, semicolon. Line 39: System dot out dot print l n, left parentheses, double quotes, back slash, t, Current stack, double quotes, right parentheses, semicolon. Line 40: for, left parentheses, Stack Walker dot Stack Frame f, colon, stack, right parentheses. Line 41: System dot out dot print l n, left parentheses, double quotes, back slash, t, back slash, t, double quotes, plus f dot get Line Number, left parentheses, right parentheses. Line 42: plus double quotes double quotes plus f dot get Method Name, left parentheses, right parentheses, right parentheses, semicolon. Line 43: blank. Line 44: return divisor, semicolon. Line 45: Right brace. Line 46: else, 2 forward slashes, general case. Line 47: Left brace. Line 48: int temp equals g c d, left parentheses, divisor, dividend percentage sign divisor, right parentheses, semicolon. Line 49: System dot out dot print l n, left parentheses, double quotes, g c d, left parentheses, double quotes, plus divisor plus double quotes, double quotes. Line 50: plus, left parentheses, dividend percentage sign divisor, right parentheses. Line 51: plus double quotes, right parentheses, returning double quotes plus temp, right parentheses, semicolon. Line 52: blank. Line 53: System dot out dot print l n, left parentheses, double quotes, back slash, t, Current stack, double quotes, right parentheses, semicolon. Line 54: for, left parentheses, Stack Walker dot Stack Frame f, colon, stack, right parentheses. Line 55: System dot out dot print l n, left parentheses, double quotes, back slash, t, back slash, t, double quotes, plus f dot get Line Number, left parentheses, right parentheses. Line 56: plus double quotes double quotes plus f dot get Method Name, left parentheses, right parentheses, right parentheses, semicolon. Line 57: blank. Line 58: return temp, semicolon. Line 59: Right brace. Line 60: Right brace. Line 61: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. Computing the number of combinations. Line 2: of picking p objects among n, using recursion. Line 3: Anderson, Franceschi. Line 4: Asterisk. Forward slash. Line 5: blank. Line 6: public class Recursive Combinations. Line 7: Left brace. Line 8: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 9: Left brace. Line 10: 2 forward slashes. compute and output number of combinations. Line 11: System dot out dot print l n, left parentheses, double quotes, C, left parentheses, 5, 2, right parentheses, equals double quotes. Line 12: plus combinations, left parentheses, 5, 2, right parentheses, right parentheses, semicolon. Line 13: Right brace. Line 14: blank. Line 15: Forward slash. Asterisk. Asterisk. recursive combinations method. Line

16: Asterisk. at symbol, param n a positive number. Line 17: Asterisk. at symbol, param p a positive number, less than or equal to n. Line 18: Asterisk. at symbol, return the number of combinations of choosing p among n. Line 19: Asterisk. Forward slash. Line 20: public static int combinations, left parentheses, int n, int p , right parentheses. Line 21: Left brace. Line 22: if, left parentheses, p equals equals 0, right parentheses, 2 forward slashes, base case, hash symbol, 1. Line 23: return 1, semicolon. Line 24: else if, left parentheses, n equals, equals, p, right parentheses, 2 forward slashes, base case hash symbol 2. Line 25: return 1, semicolon. Line 26: else, 2 forward slashes, general case. Line 27: return, left parentheses, combinations, left parentheses, n minus 1, p minus 1, right parentheses. Line 28: plus combinations, left parentheses, n minus 1, p, right parentheses, right parentheses, semicolon. Line 29: Right brace. Line 30: Right brace.

Back to Example

Line 1: public boolean recursive Palindrome, left parentheses, String p a l, right parentheses. Line 2: Left brace. Line 3: 2 forward slashes. Asterisk. Asterisk. Asterisk. Asterisk. Asterisk. Student writes the body of this method Asterisk. Asterisk. Asterisk. Asterisk. Asterisk. Line 4: 2 forward slashes. Using recursion, determine if a String representing. Line 5: 2 forward slashes, a word or a sentence is a palindrome. Line 6: 2 forward slashes, If it is, return true, otherwise return false. Line 7: 2 forward slashes. We call the animate method inside the body of this method. Line 8: 2 forward slashes. The call to animate is already coded below. Line 9: animate, left parentheses, p a l, right parentheses, , semicolon. Line 10: 2 forward slashes. Line 11: 2 forward slashes. Student code starts here. Line 12: 2 forward slashes. Line 13: return true, semicolon. 2 forward slashes. replace this dummy return statement. Line 14: 2 forward slashes. Line 15: 2 forward slashes. End of student code minus P A 1. Line 16: 2 forward slashes. Line 17: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. Demonstrating recursion on an array. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Recursion On Array. Line 6: Left brace. Line 7: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8: Left brace. Line 9: int, left square bracket, right square bracket, numbers equals Left brace, 12, 8, 10, 5, Right brace, semicolon. Line 10: int total equals sum, left parentheses, numbers, 0, right parentheses, semicolon. Line 11: System dot out dot print l n, left parentheses, double quotes, total, equals double quotes plus total, right parentheses, semicolon. Line 12: Right brace. Line 13: blank. Line 14: Forward slash. Asterisk. Asterisk. Line 15: Asterisk. sum method, colon, returns the sum of all the elements. Line 16: Asterisk. in the subarray of a r r that starts at index, index. Line 17: Asterisk. at symbol, param a r r, an array of ints. Line 18: Asterisk. at symbol, param index an int, the starting index of the subarray. Line 19: Asterisk. at symbol, return an int, the sum of the elements of the subarray. Line 20: Asterisk. Forward slash. Line 21: public static int sum, left parentheses, int, left square bracket, right square bracket, a r r, int index, right parentheses. Line 22: Left brace. Line 23: if, left parentheses, index greater than or equal to a r r dot length, right parentheses, 2 forward slashes. subarray is empty. Line 24: return 0, semicolon. Line 25: else 2 forward slashes. subarray is not empty. Line 26: return a r r, left square bracket, index, right square bracket, plus sum, left parentheses, a r r, index plus 1, right parentheses, semicolon. Line 27: Right brace. Line 28: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. Searching a sorted array using recursion. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Scanner, semicolon. Line 6: blank. Line 7: public class Recursive Binary Search. Line 8: Left brace. Line 9: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 10: Left brace. Line 11: 2 forward slashes, define an array sorted in ascending order. Line 12: int, left square bracket, right square bracket, numbers equals Left brace, 3, 6, 7, 8, 12, 15, 22, 36, 45. Line 13: 48, 51, 53, 64, 69, 72, 89, 95, Right brace, semicolon.

Line 14: blank. Line 15: Scanner scan equals new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 16: System dot out dot print, left parentheses, double quotes, Enter a value to search for greater than double quotes, right parentheses, semicolon. Line 17: int value equals scan dot next Int, left parentheses, right parentheses, semicolon. Line 18: blank. Line 19: int index equals recursive Binary Search. Line 20: left parentheses, numbers, value, 0, numbers dot length minus 1, right parentheses, semicolon. Line 21: if, left parentheses, index, exclamation mark, equals minus 1, right parentheses. Line 22: System dot out dot print l n, left parentheses, value plus double quotes found at index double quotes plus index, right parentheses, semicolon. Line 23: else. Line 24: System dot out dot print l n, left parentheses, value plus double quotes not found, double quotes, right parentheses, semicolon. Line 25: Right brace. Line 26: blank. Line 27: Forward slash. Asterisk. Asterisk. recursive Binary Search method. Line 28: Asterisk. at symbol, param a r r the array sorted in ascending order. Line 29: Asterisk. at symbol, param key the value to search for in the subarray. Line 30: Asterisk. at symbol, param start the subarray's first index. Line 31: Asterisk. at symbol, param end the subarray's last index. Line 32: Asterisk. at symbol, return the array index at which key was found. Line 33: Asterisk. or minus 1 if key was not found. Line 34: Asterisk. Forward slash. Line 35: public static int recursive Binary Search. Line 36: left parentheses, int, left square bracket, right square bracket, a r r, int key, int start, int end, right parentheses. Line 37: Left brace. Line 38: if, left parentheses, start less than equals, end, right parentheses. Line 39: Left brace. Line 40: 2 forward slashes. look at the middle element of the subarray. Line 41: int middle equals, left parentheses, start plus end , right parentheses, forward slash 2, semicolon. Line 42: blank. Line 43: if, left parentheses, a r r, left square bracket, middle, right square bracket, equals equals key, right parentheses, 2 forward slashes. found key, base case. Line 44: return middle, semicolon. Line 45: else if, left parentheses, a r r, left square bracket, middle, right square bracket, greater than key, right parentheses, 2 forward slashes. look lower. Line 46: return recursive Binary Search, left parentheses, a r r, key, start, middle minus 1, right parentheses, semicolon. Line 47: else, 2 forward slashes. look higher. Line 48: return recursive Binary Search, left parentheses, a r r, key, middle plus 1, end , right parentheses, semicolon. Line 49: Right brace. Line 50: else, 2 forward slashes. key not found, base case. Line 51: return minus 1, semicolon. Line 52: Right brace. Line 53: Right brace.

Back to Example

Line 1: public void recursive T Of H, left parentheses, int num Disks, int from Tower. Line 2: int to Tower, int use Tower, right parentheses. Line 3: Left brace. Line 4: 2 forward slashes. Asterisk. Asterisk. Asterisk. Asterisk. Asterisk. Student writes the body of this method Asterisk. Asterisk. Asterisk. Asterisk. Asterisk. Line 5: 2 forward slashes. Line 6: 2 forward slashes. Using recursion, transfer num Disks disks from the tower. Line 7: 2 forward slashes. from Tower to the tower to Tower using the tower. Line 8: 2 forward slashes. use Tower. Line 9: 2 forward slashes. The disks are numbered as follows: if we started with n disks. Line 10: 2 forward slashes. the disk at the top is disk hash symbol 1. Line 11: 2 forward slashes. and the disk at the bottom is disk hash symbol n. Line 12: 2 forward slashes. We call the move Disk method inside the body of this method. Line 13: 2 forward slashes. The move Disk method moves one disk and takes 3 arguments, colon. Line 14: 2 forward slashes. an int, representing the disk number to be moved. Line 15: 2 forward slashes. an int, representing the tower to move the disk from. Line 16: 2 forward slashes. an int, representing the tower to move the disk to. Line 17: 2 forward slashes. So if these three variables are, colon. Line 18: 2 forward slashes. disk Number, from Tower, and to Tower. Line 19: 2 forward slashes. then the call to move Disks will be, colon. Line 20: 2 forward slashes. move Disk, left parentheses, disk Number, from Tower, to Tower, right parentheses, semicolon. Line 21: if, left parentheses, num Disks greater than 0, right parentheses. Line 22: Left brace. Line 23: 2 forward slashes. Student code starts here. Line 24: 2 forward slashes. 1. Move, left parentheses, num Disks minus 1, right parentheses, disks from from Tower. Line 25: 2 forward slashes. to use Tower using to Tower. Line 26: 2 forward slashes. 2. Move one disk from from Tower to to

Tower. Line 27: 2 forward slashes. Print a message to the screen, then. Line 28: 2 forward slashes. call move Disk in order to animate. Line 29: 2 forward slashes. 3. Move, left parentheses, num Disks minus 1, right parentheses, disks from use Tower to to Tower. Line 30: 2 forward slashes. using from Tower. Line 31: Right brace. Line 32: 2 forward slashes. Base case, colon, 0 disks to move equals equals greater than do nothing. Line 33: 2 forward slashes. Line 34: 2 forward slashes. end of student code. Line 35: 2 forward slashes. Line 36: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. Sorting an array using merge sort. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Merge Sort. Line 6: Left brace. Line 7: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8: Left brace. Line 9: int, left square bracket, right square bracket, numbers equals Left brace, 66, 54, 75, 23, 15, 10, 98, 80, Right brace, semicolon. Line 10: merge Sort, left parentheses, numbers, 0, numbers dot length minus 1, right parentheses, semicolon. Line 11: for, left parentheses, int i equals 0, semicolon, i less than numbers dot length, semicolon, i, plus, plus, right parentheses. Line 12: System dot out dot print, left parentheses, numbers, left square bracket, i, right square bracket, plus double quotes double quotes, right parentheses, semicolon. Line 13: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 14: Right brace. Line 15: blank. Line 16: Forward slash. Asterisk. Asterisk. Line 17: Asterisk. merge Sort method, colon, sorts the subarray of a r r. Line 18: Asterisk, left parentheses, from index start to index end, both included, right parentheses. Line 19: Asterisk, in ascending order. Line 20: Asterisk, at symbol, param a r r, an array of ints. Line 21: Asterisk, at symbol, param start an int, the starting index of the subarray. Line 22: Asterisk, at symbol, param end an int, the ending index of the subarray. Line 23: Asterisk. Forward slash. Line 24: public static void merge Sort, left parentheses, int, left square bracket, right square bracket, a r r, int start, int end, right parentheses. Line 25: Left brace. Line 26: if, left parentheses, start less than end, right parentheses, 2 forward slashes, general case. Line 27: Left brace. Line 28: int middle equals, left parentheses, start plus end, right parentheses, forward slash 2, semicolon. Line 29: 2 forward slashes. sort the left part of the array. Line 30: merge Sort, left parentheses, a r r, start, middle, right parentheses, semicolon. Line 31: 2 forward slashes, sort the right part of the array. Line 32: merge Sort, left parentheses, a r r, middle plus 1, end, right parentheses, semicolon. Line 33: 2 forward slashes, merge the 2 parts. Line 34: merge, left parentheses, a r r, start, middle, middle plus 1, end, right parentheses, semicolon. Line 35: Right brace. Line 36: 2 forward slashes. else, base case, empty or 1 element array, already sorted. Line 37: Right brace. Line 38: blank. Line 39: Forward slash. Asterisk.

Asterisk. Line 40: Asterisk. merge method, colon, merges, in ascending order. Line 41: Asterisk. the two subarrays of a r r defined as follows, colon. Line 42: Asterisk. left subarray, colon, between indexes start 1 and end 1, both included. Line 43: Asterisk. right subarray, colon, between indexes start 2 and end 2, both included. Line 44: Asterisk. at symbol, param a r r, an array of ints. Line 45: Asterisk. at symbol, param start 1 an int, the starting index of the left subarray. Line 46: Asterisk. at symbol, param end 1 an int, the ending index of the left subarray. Line 47: Asterisk. at symbol, param start 2 an int, the starting index of the right subarray. Line 48: Asterisk. at symbol, param end 2 an int, the ending index of the right subarray. Line 49: Asterisk. Forward slash. Line 50: public static void merge, left parentheses, int, left square bracket, right square bracket, a r r, int start 1, int end 1. Line 51: int start 2, int end 2, right parentheses. Line 52: Left brace. Line 53: 2 forward slashes, merge the 2 sides into a temp array. Line 54: int, left square bracket, right square bracket, temp equals new int, left square bracket, a r r dot length, right square bracket, semicolon. 2 forward slashes, temporary array. Line 55: int i equals start 1, semicolon. Line 56: int j equals start 2, semicolon. Line 57: int k equals start 1, semicolon. Line 58: blank. Line 59: 2 forward slashes. generate merged subarray until one side is done. Line 60: while, left parentheses, i less than equals end 1 2 ampersands j less than equals end 2, right parentheses. Line 61: Left brace. Line 62: if, left parentheses, a r r, left square bracket, i, right square bracket, less than a r r, left square bracket, j, right square bracket, right parentheses. Line 63: Left brace. Line 64: temp, left square bracket, k, right square bracket, equals a r r, left square bracket, i, right square bracket, semicolon. Line 65: i, plus, plus, semicolon. Line 66: Right brace. Line 67: else. Line 68: Left brace. Line 69: temp, left square bracket, k, right square bracket, equals a r r, left square bracket, j, right square bracket, , semicolon. Line 70: j, plus, plus, semicolon. Line 71: Right brace. Line 72: k, plus, plus, semicolon. Line 73: Right brace. Line 74: blank. Line 75: 2 forward slashes. flush left subarray if necessary. Line 76: while, left parentheses, i less than equals, end 1, right parentheses. Line 77: Left brace. Line 78: temp, left square bracket, k, right square bracket, equals a r r, left square bracket, i, right square bracket, semicolon. Line 79: i, plus, plus, semicolon. Line 80: k, plus, plus, semicolon. Line 81: Right brace. Line 82: blank. Line 83: 2 forward slashes. flush right subarray if necessary. Line 84: while, left parentheses, j less than equals end2, right parentheses. Line 85: Left brace. Line 86: temp, left square bracket, k, right square bracket, equals a r r, left square bracket, j, right square bracket, semicolon. Line 87: j, plus, plus, semicolon. Line 88: k, plus, plus, semicolon. Line 89: Right brace. Line 90: blank. Line 91: 2 forward slashes, write back temp into a r r. Line 92: for, left parentheses, i equals start 1, semicolon. i less than equals end 2, semicolon. i, plus, plus, right parentheses. Line 93: a r r, left square bracket, i, right square bracket, equals temp, left square bracket, i, right square bracket, semicolon. Line 94: Right brace. Line 95: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. Sorting an array using quick sort. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Quick Sort. Line 6: Left brace. Line 7: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8: Left brace. Line 9: int, left square bracket, right square bracket, numbers equals Left brace, 66, 54, 75, 23, 15, 10, 98, 80 Right brace, semicolon. Line 10: quick Sort, left parentheses, numbers, 0, numbers dot length minus 1, right parentheses, semicolon. Line 11: for, left parentheses, int i equals 0, semicolon, i less than numbers dot length, semicolon, i, plus, plus, right parentheses. Line 12: System dot out dot print, left parentheses, numbers, left square bracket, i, right square bracket, plus double quotes double quotes, right parentheses, semicolon. Line 13: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 14: Right brace. Line 15: blank. Line 16: Forward slash. Asterisk. Asterisk. Line 17: Asterisk. quick Sort method, colon, sorts the subarray of a r r. Line 18: Asterisk. left parentheses, from index start to index end, both included, right parentheses. Line 19: Asterisk. in ascending order. Line 20: Asterisk. at symbol, param a r r, an array of ints. Line 21: Asterisk. at symbol, param start an int, the starting index of the subarray. Line

22: Asterisk. at symbol, param end an int, the ending index of the subarray. Line 23: Asterisk. Forward slash. Line 24: public static void quick Sort, left parentheses, int, left square bracket, right square bracket, a r r, int start, int end, right parentheses. Line 25: Left brace. Line 26: if, left parentheses, end greater than start, right parentheses, 2 forward slashes. general case, greater than or equals 2 elements. Line 27: Left brace. Line 28: int pivot Index equals partition, left parentheses, a r r, start, end, right parentheses, semicolon. Line 29: 2 forward slashes. quick sort left subarray. Line 30: quick Sort, left parentheses, a r r, start, pivot Index minus 1, right parentheses, semicolon. Line 31: 2 forward slashes. quick sort right subarray. Line 32: quick Sort, left parentheses, a r r, pivot Index plus 1, end, right parentheses, semicolon. Line 33: Right brace. Line 34: 2 forward slashes. else, 0 or 1 element equals equals greater than subarray is sorted, do nothing. Line 35: Right brace. Line 36: blank. Line 37: Forward slash. Asterisk. Asterisk. Line 38: Asterisk. partition method, colon, partitions the subarray of a r r. Line 39: Asterisk. left parentheses, from index start to index end, both included, right parentheses. Line 40: Asterisk. around the pivot, a r r, left square bracket, start, right square bracket. Line 41: Asterisk. at symbol, param a r r, an array of ints. Line 42: Asterisk. at symbol, param start an int, the starting index of the subarray. Line 43: Asterisk. at symbol, param end an int, the ending index of the subarray. Line 44: Asterisk. Forward slash. Line 45: public static int partition, left parentheses, int, left square bracket, right square bracket, a r r, int start, int end, right parentheses. Line 46: Left brace. Line 47: int j equals start, semicolon. Line 48: int pivot equals a r r, left square bracket, start, right square bracket, semicolon. Line 49: for, left parentheses, int i equals start plus 1, semicolon, i less than equals end, semicolon, i, plus, plus, right parentheses. Line 50: Left brace. Line 51: if, left parentheses, a r r, left square bracket, i, right square bracket, less than pivot, right parentheses. Line 52: Left brace. Line 53: j, plus, plus, semicolon. Line 54: swap, left parentheses, a r r, i, j, right parentheses, semicolon. Line 55: Right brace. Line 56: Right brace. Line 57: swap, left parentheses, a r r, start, j, right parentheses, semicolon. Line 58: return j, semicolon. Line 59: Right brace. Line 60: blank. Line 61: Forward slash. Asterisk. Asterisk. Line 62: Asterisk. swap method, colon, swaps the elements of a r r. Line 63: Asterisk. at index i and index j. Line 64: Asterisk. at symbol, param a r r, an a r ray of ints. Line 65: Asterisk. at symbol, param i an int, a valid index of a r r. Line 66: Asterisk. at symbol, param j an int, a valid index of a r r. Line 67: Asterisk. Forward slash. Line 68: public static void swap, left parentheses, int, left square bracket, right square bracket, a r r, int i, int j, right parentheses. Line 69: Left brace. Line 70: int temp equals a r r, left square bracket, i, right square bracket, semicolon. Line 71: a r r, left square bracket, i, right square bracket, equals a r r, left square bracket, j, right square bracket, semicolon. Line 72: a r r, left square bracket, j, right square bracket, equals temp, semicolon. Line 73: Right

brace. Line 74: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. The Integer Node class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Integer Node. Line 6: Left brace. Line 7: private int data semicolon. Line 8: private Integer Node next semicolon. Line 9: blank. Line 10: Forward slash. Asterisk. Asterisk default constructor. Line 11: Asterisk sets data to 0 comma and next to null. Line 12: Asterisk. Forward slash. Line 13: public Integer Node Left parentheses, right parentheses. Line 14: Left brace. Line 15: data equals 0 semicolon. Line 16: next equals null semicolon. Line 17: Right brace. Line 18: blank. Line 19: Forward slash. Asterisk. Asterisk overloaded constructor. Line 20: Asterisk at symbol param data data value. Line 21: Asterisk. Forward slash. Line 22: public Integer Node left parentheses int data right parentheses. Line 23: Left brace. Line 24: set Data left parentheses data right parentheses semicolon. Line 25: next equals null semicolon. Line 26: Right brace. Line 27: blank. Line 28: Forward slash. Asterisk. Asterisk accessor for data. Line 29: Asterisk at symbol return the value of the node. Line 30: Asterisk. Forward slash. Line 31: public int get Data Left parentheses, right parentheses. Line 32: Left brace. Line 33: return data semicolon. Line 34: Right brace. Line 35: blank. Line 36: Forward slash. Asterisk. Asterisk accessor for next. Line 37: Asterisk at symbol return the reference to the next node. Line 38: Asterisk. Forward slash. Line 39: public Integer Node get Next Left parentheses, right parentheses. Line 40: Left brace. Line 41: return next semicolon. Line 42: Right brace. Line 43: blank. Line 44: Forward slash. Asterisk. Asterisk mutator for data. Line 45: Asterisk at symbol param data the new value for the node. Line 46: Asterisk at symbol return a reference to this object. Line 47: Asterisk. Forward slash. Line 48: public Integer Node set Data left parentheses int data right parentheses. Line 49: Left brace. Line 50: this dot data equals data semicolon. Line 51: return this semicolon. Line 52: Right brace. Line 53: blank. Line 54: Forward slash. Asterisk. Asterisk mutator for next. Line 55: Asterisk at symbol param next the new value for next. Line 56: Asterisk at symbol return a reference to this object. Line 57: Asterisk. Forward slash. Line 58: public Integer Node set Next left parentheses Integer Node next right parentheses. Line 59: Left brace. Line 60: this dot next equals next semicolon. Line 61: return this semicolon. Line 62: Right brace. Line 63: Right brace.

Back to Example

The table is titled Methods of the Integer Linked List class. The table has 2 columns labeled Return Value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: insert, left parentheses, int value, right parentheses, inserts value at the beginning of the list. Row 2. Return value: boolean. Method name and argument list: delete, left parentheses, int value, right parentheses, removes the first item on the list that is equal to value and returns true. If there is no such item on the list, the method returns false. Row 3. Return value: String. Method name and argument list: to String, left parentheses, right parentheses, returns a String representation of the list.

Back to Table

Line 1: Forward slash. Asterisk. The Integer Linked List class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Integer Linked List. Line 6: Left brace. Line 7: private Integer Node head semicolon. Line 8: private int number Of Items semicolon. Line 9: blank. Line 10: Forward slash. Asterisk. Asterisk default constructor. Line 11: Asterisk constructs an empty list. Line 12: Asterisk. Forward slash. Line 13: public Integer Linked List Left parentheses, right parentheses. Line 14: Left brace. Line 15: head equals null semicolon. Line 16: number Of Items equals 0 semicolon. Line 17: Right brace. Line 18: blank. Line 19: Forward slash. Asterisk. Asterisk accessor for number Of Items. Line 20: Asterisk at symbol return number Of Items. Line 21: Asterisk. Forward slash. Line 22: public int get Number Of Items Left parentheses, right parentheses. Line 23: Left brace. Line 24: return number Of Items semicolon. Line 25: Right brace. Line 26: blank. Line 27: Forward slash. Asterisk. Asterisk insert method. Line 28: Asterisk at symbol param value data to insert. Line 29:

Asterisk inserts node at head. Line 30: Asterisk. Forward slash. Line 31: public void insert left parentheses int value right parentheses. Line 32: Left brace. Line 33: Integer Node n d equals new Integer Node left parentheses value right parentheses semicolon. Line 34: n d dot set Next left parentheses head right parentheses semicolon. Line 35: head equals n d semicolon. Line 36: number Of Items plus plus semicolon. Line 37: Right brace. Line 38: blank. Line 39: Forward slash. Asterisk. Asterisk delete method. Line 40: Asterisk at symbol param value the value to delete. Line 41: Asterisk at symbol return true if value was deleted from the list comma false otherwise. Line 42: Asterisk. Forward slash. Line 43: public boolean delete left parentheses int value right parentheses. Line 44: Left brace. Line 45: Integer Node current equals head semicolon. Line 46: Integer Node previous equals null semicolon. Line 47: while left parentheses current exclamation mark equals null. Line 48: 2 ampersands current dot get Data Left parentheses, right parentheses exclamation mark equals value right parentheses. Line 49: Left brace. Line 50: previous equals current semicolon. Line 51: current equals current dot get Next Left parentheses, right parentheses semicolon. Line 52: Right brace. Line 53: blank. Line 54: if left parentheses current equals equals null right parentheses 2 forward slashes not found. Line 55: return false semicolon. Line 56: else. Line 57: Left brace. Line 58: if left parentheses current equals equals head right parentheses. Line 59: head equals head dot get Next Left parentheses, right parentheses semicolon 2 forward slashes delete head. Line 60: else. Line 61: previous dot set Next left parentheses current dot get Next Left parentheses, right parentheses right parentheses semicolon. Line 62: blank. Line 63: number Of Items minus minus semicolon. Line 64: return true semicolon. Line 65: Right brace. Line 66: Right brace. Line 67: blank. Line 68: Forward slash. Asterisk. Asterisk to String. Line 69: Asterisk at symbol return values in list separated by a space. Line 70: Asterisk. Forward slash. Line 71: at symbol Override. Line 72: public String to String Left parentheses, right parentheses. Line 73: Left brace. Line 74: String list String equals double quotes double quotes semicolon. Line 75: Integer Node current equals head semicolon. Line 76: for left parentheses int i equals 0 semicolon i less than number Of Items semicolon i plus plus right parentheses. Line 77: Left brace. Line 78: list String plus equals current dot get Data Left parentheses, right parentheses plus double quotes double quotes semicolon. Line 79: current equals current dot get Next Left parentheses, right parentheses semicolon. Line 80: Right brace. Line 81: return list String semicolon. Line 82: Right brace. Line 83: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. The Integer Linked List Test class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Integer Linked List Test. Line 6: Left brace. Line 7: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8: Left brace. Line 9: 2 forward slashes construct empty Integer Linked List. Line 10: Integer Linked List numbers equals new Integer Linked List Left parentheses, right parentheses, semicolon. Line 11: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 12: plus numbers dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus, numbers dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 13: blank. Line 14: numbers dot insert, left parentheses, 7, right parentheses, semicolon, 2 forward slashes, insert in empty list. Line 15: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 16: plus numbers dot get Number Of Items, Left parentheses, right parentheses, plus double quotes, backslash n, double quotes plus numbers dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 17: blank. Line 18: numbers dot insert, left parentheses, 2 right parentheses, semicolon, 2 forward slashes, insert in list with one item. Line 19: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 20: plus numbers dot get Number Of Items, Left parentheses, right parentheses, plus double quotes, backslash n, double quotes, plus numbers dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 21: blank. Line 22: numbers dot insert, left parentheses, 5, right parentheses, semicolon, 2 forward slashes, insert in list with two items. Line 23: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 24: plus numbers dot get Number Of Items, Left parentheses, right parentheses, plus double quotes, backslash n, double quotes, plus, numbers dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 25: blank. Line 26: if left parentheses, exclamation mark, numbers dot delete, left parentheses, 8, right parentheses, right parentheses, 2 forward slashes, unsuccessful minus not in list. Line 27: System dot out dot print l n, left parentheses, double quotes, 8, could not be deleted, colon, double quotes, right parentheses, semicolon. Line 28: blank. Line 29: if left parentheses numbers dot delete, left parentheses, 2, right parentheses, right parentheses, 2 forward slashes, successful. Line 30: System dot out dot print l n, left parentheses, double quotes, 2 was successfully deleted, colon, double quotes, right parentheses, semicolon. Line 31: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 32: plus numbers dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus, numbers dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 33: blank. Line 34: if, left parentheses, numbers dot delete, left parentheses, 7, right parentheses, right parentheses, 2 forward slashes, successful. Line 35: System dot out dot print l n, left parentheses, double quotes, 7 was successfully deleted, colon, double quotes, right parentheses, semicolon. Line 36: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 37: plus numbers dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus, numbers dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 38: blank. Line 39: if left parentheses, numbers dot delete, left parentheses, 5, right parentheses, right parentheses, 2 forward slashes, successful. Line 40: System dot out dot print l n, left parentheses, double quotes, 5 was successfully deleted, colon, double quotes, right parentheses, semicolon. Line 41: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 42: plus, numbers dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus, numbers dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 43: blank. Line 44: if left parentheses, exclamation mark, numbers dot delete, left parentheses, 8, right parentheses, right parentheses, 2 forward slashes, unsuccessful minus empty list. Line 45: System dot out dot print l n, left parentheses, double quotes, 8 could not be deleted, colon, double quotes, right parentheses, semicolon. Line 46: System dot out dot print I n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 47: plus numbers dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus, numbers dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 48: Right brace. Line 49: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. The Player Class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Player. Line 6: Left brace. Line 7: private int i d semicolon. Line 8: private String name semicolon. Line 9: private String game semicolon. Line 10: blank. Line 11: Forward slash. Asterisk. Asterisk constructor. Line 12: Asterisk at symbol param i d player's i d. Line 13: Asterisk at symbol param name player's name. Line 14: Asterisk at symbol param game player's game. Line 15: Asterisk. Forward slash. Line 16: public Player, left parentheses, int i d, comma, String name, comma, String game, right parentheses. Line 17: Left brace. Line 18: set I D, left parentheses, i d, right parentheses, semicolon. Line 19: set Name left parentheses, name, right parentheses, semicolon. Line 20: set Game, left parentheses, game, right parentheses, semicolon. Line 21: Right brace. Line 22: blank. Line 23: Forward slash. Asterisk. Asterisk accessor for i d. Line 24: Asterisk at symbol return i d. Line 25: Asterisk. Forward slash. Line 26: public int get I D, Left parentheses, right parentheses. Line 27: Left brace. Line 28: return i d, semicolon. Line 29: Right brace. Line 30: blank. Line 31: Forward slash. Asterisk. Asterisk accessor for name. Line 32: Asterisk at symbol return name. Line 33: Asterisk. Forward slash. Line 34: public String get Name, Left parentheses, right parentheses. Line 35: Left brace. Line 36: return name, semicolon. Line 37: Right brace. Line 38: blank. Line 39: Forward slash. Asterisk. Asterisk accessor for game. Line 40: Asterisk at symbol return game. Line 41: Asterisk. Forward slash. Line 42: public String get Game Left parentheses, right parentheses. Line 43: Left brace. Line 44: return game semicolon. Line 45: Right brace. Line 46: blank. Line 47: Forward slash. Asterisk. Asterisk mutator for i d. Line 48: Asterisk at symbol param i d new value for i d. Line 49: Asterisk at symbol return a reference to this object. Line 50: Asterisk. Forward slash. Line 51: public Player set I D, left parentheses, int i d, right parentheses. Line 52: Left brace. Line 53: this dot i d equals i d, semicolon. Line 54: return this, semicolon. Line 55: Right brace. Line 56: blank. Line 57: Forward slash. Asterisk. Asterisk mutator for name. Line 58: Asterisk at symbol param name new value for name. Line 59: Asterisk at symbol return a reference to this object. Line 60: Asterisk. Forward slash. Line 61: public Player set Name, left parentheses, String, name, right parentheses. Line 62: Left brace. Line 63: this dot name equals name, semicolon. Line 64: return this, semicolon. Line 65: Right brace. Line 66: blank. Line 67: Forward slash. Asterisk.

Asterisk mutator for game. Line 68: Asterisk at symbol param game new value for game. Line 69: Asterisk at symbol return a reference to this object. Line 70: Asterisk. Forward slash. Line 71: public Player set Game, left parentheses, String game, right parentheses. Line 72: Left brace. Line 73: this dot game equals game, semicolon. Line 74: return this, semicolon. Line 75: Right brace. Line 76: blank. Line 77: Forward slash. Asterisk. Asterisk equals method. Line 78: Asterisk at symbol param o reference to object to compare to this object. Line 79: Asterisk at symbol return true if o is a Player object. Line 80: Asterisk and i d comma name comma and game are equal in both objects semicolon false otherwise. Line 81: Asterisk. Forward slash. Line 82: at symbol Override. Line 83: public boolean equals, left parentheses, Object o right parentheses. Line 84: Left brace. Line 85: if left parentheses exclamation mark, left parentheses o instance of Player right parentheses right parentheses. Line 86: return false, semicolon. Line 87: else. Line 88: Left brace. Line 89: Player o b j Player equals left parentheses Player right parentheses o, semicolon. Line 90: return left parentheses i d equals equals o b j Player dot i d 2 ampersands name dot equals, left parentheses, o b j Player dot name right parentheses. Line 91: 2 ampersands game dot equals left parentheses o b j Player dot game right parentheses right parentheses, semicolon. Line 92: Right brace. Line 93: Right brace. Line 94: blank. Line 95: Forward slash. Asterisk. Asterisk to String method. Line 96: Asterisk at symbol return String representation of Player object. Line 97: Asterisk. Forward slash. Line 98: at symbol Override. Line 99: public String to String, Left parentheses, right parentheses. Line 100: Left brace. Line 101: return, left parentheses, double quotes i d, colon, double quotes, plus, i d plus, double quotes, backslash t, name, colon, double quotes. Line 102: plus name plus double quotes, backslash t, game, colon, double quotes plus game, right parentheses, semicolon. Line 103: Right brace. Line 104: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. The Player Node class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Player Node. Line 6: Left brace. Line 7: private Player player, semicolon. Line 8: private Player Node next, semicolon. Line 9: Forward slash. Asterisk. Asterisk default constructor. Line 10: Asterisk initializes player and next references to null. Line 11: Asterisk. Forward slash. Line 12: public Player Node, Left parentheses, right parentheses. Line 13: Left brace. Line 14: player equals null, semicolon. Line 15: next equals null, semicolon. Line 16: Right brace. Line 17: blank. Line 18: Forward slash. Asterisk. Asterisk overloaded constructor. Line 19: Asterisk at symbol param player. Line 20: Asterisk initializes player reference to player reference to Player object. Line 21: Asterisk. Forward slash. Line 22: public Player Node left parentheses Player player right parentheses. Line 23: Left brace. Line 24: set Player, left parentheses, player right parentheses, semicolon. Line 25: next equals null, semicolon. Line 26: Right brace. Line 27: blank. Line 28: Forward slash. Asterisk. Asterisk accessor for player. Line 29: Asterisk at symbol return player. Line 30: Asterisk. Forward slash. Line 31: public Player get Player Left parentheses, right parentheses. Line 32: Left brace. Line 33: return player semicolon. Line 34: Right brace. Line 35: blank. Line 36: Forward slash. Asterisk. Asterisk accessor for next. Line 37: Asterisk at symbol return next. Line 38: Asterisk. Forward slash. Line 39: public Player Node get Next Left parentheses, right parentheses. Line 40: Left brace. Line 41: return next, semicolon. Line 42: Right brace. Line 43: blank. Line 44: Forward slash. Asterisk. Asterisk mutator for player. Line 45: Asterisk at symbol param player new Player reference. Line 46: Asterisk at symbol return a reference to this object. Line 47: Asterisk. Forward slash. Line 48: public Player Node set Player, left parentheses, Player player right parentheses. Line 49: Left brace. Line 50: this dot player equals player, semicolon. Line 51: return this, semicolon. Line 52: Right brace. Line 53: blank. Line 54: Forward slash. Asterisk. Asterisk mutator for next. Line 55: Asterisk at symbol param next new reference to next Player Node. Line 56: Asterisk at symbol return a reference to this object. Line 57: Asterisk. Forward slash. Line 58: public Player Node set Next left parentheses Player Node next right parentheses. Line 59: Left brace. Line 60: this dot next equals next, semicolon. Line 61:

return this, semicolon. Line 62: Right brace. Line 63: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. The Shell Linked List class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public abstract class Shell Linked List. Line 6: Left brace. Line 7: protected Player Node head semicolon. Line 8: protected int number Of Items semicolon. Line 9: blank. Line 10: Forward slash. Asterisk. Asterisk constructor. Line 11: Asterisk sets head to null and number Of Items to 0. Line 12: Asterisk. Forward slash. Line 13: public Shell Linked List, Left parentheses, right parentheses. Line 14: Left brace. Line 15: head equals null, semicolon. Line 16: number Of Items equals 0 semicolon. Line 17: Right brace. Line 18: blank. Line 19: Forward slash. Asterisk. Asterisk accessor for number Of Items. Line 20: Asterisk at symbol return number Of Items. Line 21: Asterisk. Forward slash. Line 22: public int get Number Of Items, Left parentheses, right parentheses. Line 23: Left brace. Line 24: return number Of Items semicolon. Line 25: Right brace. Line 26: blank. Line 27: Forward slash. Asterisk. Asterisk is Empty method. Line 28: Asterisk at symbol return true if no items in list semicolon false otherwise. Line 29: Asterisk. Forward slash. Line 30: public boolean is Empty Left parentheses, right parentheses. Line 31: Left brace. Line 32: return left parentheses number Of Items equals equals 0 right parentheses semicolon. Line

33: Right brace. Line 34: blank. Line 35: Forward slash. Asterisk. Asterisk to String method. Line 36: Asterisk at symbol return the contents of the list. Line 37: Asterisk. Forward slash. Line 38: at symbol Override. Line 39: public String to String, Left parentheses, right parentheses. Line 40: Left brace. Line 41: String list String equals double quotes, double quotes, semicolon. Line 42: Player Node current equals head semicolon. Line 43: while left parentheses current exclamation mark, equals null right parentheses. Line 44: Left brace. Line 45: list String plus equals current dot get Player, Left parentheses, right parentheses dot to String Left Line parentheses, right parentheses plus double quotes, backslash n, double quotes, semicolon. Line 46: current equals current dot get Next, Left parentheses, right parentheses, semicolon. Line 47: Right brace. Line 48: return list String, semicolon. Line 49: Right brace. Line 50: Right brace.

Back to Example

The table is titled, Constructor and Methods of the abstract Shell Linked List Class. The columns of the first row are labeled Class, Constructor and argument list. The columns of the remaining 3 rows are labeled Return Value and Method name and argument list. The row entries are as follows. Row 1. Class: Shell Linked List. Constructor and argument list: Shell Linked List, left parentheses, right parentheses, constructs an empty list. Row 2. Return value: int. Method name and argument list: get Number Of

Items, left parentheses, right parentheses, returns the number of items in the list. Row 3. Return value: boolean. Method name and argument list: is Empty, left parentheses, right parentheses, returns true if the list contains 0 items, false otherwise. Row 4. Return value: String. Method name and argument list: to String, left parentheses, right parentheses, returns the contents of every node in the list.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, String. Line 1: Forward slash, asterisk. The Data Structure Exception class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: import java dot util dot No Such Element Exception, semicolon. Line 5: blank. Line 6: public class Data Structure Exception extends No Such Element Exception. Line 7: left brace. Line 8, indented once: Forward slash, 2 asterisks, constructor. Line 9, indented once: asterisk, at symbol, param s error message. Line 10, indented once: Asterisk, Forward slash. Line 11, indented once: public Data Structure Exception, left parentheses, String s, right parentheses. Line 12, indented once: left brace. Line 13, indented twice: super, left parentheses, s, right parentheses, semicolon. Line 14, indented once: right brace. Line 15: right brace.

Back to Example

The table is titled Methods of the Player Linked List Class. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: insert, left parentheses, Player p, right parentheses, inserts Player p at the beginning of the list. Row 2. Return value: Player. Method name and argument list: delete, left parentheses, int search I D, right parentheses, returns and removes the first Player in the list with an I D equal to search I D. If there is no such Player in the list, the method throws a Data Structure Exception. Row 3. Return value: Player. Method name and argument list: peek, left parentheses, int search I D, right parentheses, returns the first Player on the list whose I D is equal to search I D. If there is no such Player in the list, the method throws a Data Structure Exception.

Back to Table

Line 1: Forward slash. Asterisk. The Player Linked List class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Player Linked List extends Shell Linked List. Line 6: Left brace. Line 7: blank. Line 8: Forward slash. Asterisk. Asterisk default constructor. Line 9: Asterisk calls constructor of Shell Linked List class. Line 10: Asterisk. Forward slash. Line 11: public Player Linked List Left parentheses, right parentheses. Line 12: Left brace. Line 13: super, Left parentheses, right parentheses, semicolon. Line 14: Right brace. Line 15: blank. Line 16:

Forward slash. Asterisk. Asterisk insert method. Line 17: Asterisk at symbol param p Player object to insert. Line 18: Asterisk. Forward slash. Line 19: public void insert, left parentheses, Player p right parentheses. Line 20: Left brace. Line 21: 2 forward slashes insert as head. Line 22: Player Node p n equals new Player Node, left parentheses, p right parentheses, semicolon. Line 23: p n dot set Next, left parentheses, head, right parentheses, semicolon. Line 24: head equals p n semicolon. Line 25: number Of Items plus plus semicolon. Line 26: Right brace. Line 27: blank. Line 28: Forward slash. Asterisk. Asterisk delete method. Line 29: Asterisk at symbol param search I D i d of Player to delete. Line 30: Asterisk at symbol return the Player deleted. Line 31: Asterisk. Forward slash. Line 32: public Player delete left parentheses int search I D right parentheses. Line 33: throws Data Structure Exception. Line 34: Left brace. Line 35: Player Node current equals head semicolon. Line 36: Player Node previous equals null semicolon. Line 37: while left parentheses current exclamation mark, equals null. Line 38: 2 ampersands, current dot get Player Left parentheses, right parentheses dot get I D Left parentheses, right parentheses, exclamation mark, equals search I D right parentheses. Line 39: Left brace. Line 40: previous equals current semicolon. Line 41: current equals current dot get Next Left parentheses, right parentheses, semicolon. Line 42: Right brace. Line 43: blank. Line 44: if left parentheses current equals equals null right parentheses 2 forward slashes not found. Line

45: throw new Data Structure Exception left parentheses search I D. Line 46: plus double quotes not found, colon, cannot be deleted double quotes right parentheses semicolon. Line 47: else. Line 48: Left brace. Line 49: if left parentheses current equals equals head right parentheses. Line 50: head equals head dot get Next Left parentheses, right parentheses semicolon 2 forward slashes delete head. Line 51: else. Line 52: previous dot set Next left parentheses current dot get Next Left parentheses, right parentheses right parentheses semicolon. Line 53: blank. Line 54: number Of Items minus minus semicolon. Line 55: return current dot get Player Left parentheses, right parentheses semicolon. Line 56: Right brace. Line 57: Right brace. Line 58: blank. Line 59: Forward slash. Asterisk. Asterisk peek method. Line 60: Asterisk at symbol param search I D i d of Player to search for. Line 61: Asterisk at symbol return the Player found. Line 62: Asterisk. Forward slash. Line 63: public Player peek left parentheses int search I D right parentheses. Line 64: throws Data Structure Exception. Line 65: Left brace. Line 66: Player Node current equals head semicolon. Line 67: while left parentheses current exclamation mark, equals null. Line 68: 2 ampersands current dot get Player Left parentheses, right parentheses dot get I D Left parentheses, right parentheses exclamation mark, equals search I D right parentheses. Line 69: Left brace. Line 70: current equals current dot get Next Left parentheses, right parentheses semicolon. Line 71: Right brace. Line 72: blank. Line 73: if left parentheses current equals equals null right parentheses 2 forward slashes not found. Line 74: throw new Data Structure Exception, left parentheses search I D. Line 75: plus double quotes not found, colon, cannot be deleted double quotes right parentheses semicolon. Line 76: else. Line 77: Left brace. Line 78: return current dot get Player Left parentheses, right parentheses semicolon. Line 79: Right brace. Line 80: Right brace. Line 81: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. The Player Linked List Test class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Player Linked List Test. Line 6: Left brace. Line 7: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 8: Left brace. Line 9: Player p 1 equals new Player, left parentheses, 7 comma, double quotes, Sarah, double quotes, comma, double quotes, Mario, double quotes, right parentheses, semicolon. Line 10: Player p 2 equals new Player, left parentheses, 2, comma, double quotes, Jin, double quotes, comma, double quotes, Golf, double quotes, right parentheses, semicolon. Line 11: Player p 3 equals new Player, left parentheses, 5, comma, double quotes, Ajay, double quotes, comma, double quotes, Sonic, double quotes right parentheses, semicolon. Line 12: blank. Line 13: 2 forward slashes construct empty Player Linked List. Line 14: Player Linked List players equals new Player Linked List, Left parentheses, right parentheses, semicolon. Line 15: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 16: plus players dot get Number Of Items, Left parentheses, right parentheses, plus double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 17: blank. Line 18: players dot insert, left parentheses, p 1, right parentheses, semicolon, 2 forward slashes, insert in empty list. Line 19: System dot out dot print l n left parentheses double quotes, Number of items in the list, colon, double quotes. Line 20: plus players dot get Number Of Items, Left parentheses, right parentheses plus double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 21: blank. Line 22: players dot insert, left parentheses, p 2, right parentheses, semicolon, 2 forward slashes, insert in list of one item. Line 23: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 24: plus players dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 25: blank. Line 26: players dot insert, left parentheses, p 3, right parentheses, semicolon, 2 forward slashes, insert in list of two items. Line 27: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 28: plus players dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 29: blank. Line 30: Player temp, semicolon, 2 forward slashes, will be assigned the deleted item. Line 31: blank. Line 32: try. Line 33: Left brace. Line 34: temp equals players dot delete, left parentheses, 8, right parentheses, semicolon, 2 forward slashes, unsuccessful. Line 35: System dot out dot print l n, left parentheses, double quotes, Player deleted, colon, double quotes, plus temp, right parentheses, semicolon. Line 36: Right brace. Line 37: catch, left parentheses, Data Structure Exception, d s e 1, right parentheses. Line 38: Left brace. Line 39: System dot out dot print l n, left parentheses, d s e 1 dot get Message, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, right parentheses, semicolon. Line 40: Right brace. Line 41: blank. Line 42: try. Line 43: Left brace. Line 44: temp equals players dot peek, left parentheses, 2 right parentheses, semicolon, 2 forward slashes, test peek. Line 45: System dot out dot print l n, left parentheses, double quotes, Player retrieved, colon, double quotes, plus temp, right parentheses, semicolon. Line 46: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 47: plus players dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 48: blank. Line 49: temp equals players dot delete, left parentheses, 2 right parentheses, semicolon, 2 forward slashes, delete in the middle. Line 50: System dot out dot print l n, left parentheses, double quotes, Player deleted, colon, double quotes, plus temp right parentheses, semicolon. Line 51: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 52: plus players dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus, players dot to String, Left parentheses, right parentheses right parentheses, semicolon. Line 53: blank. Line 54: temp equals players dot delete, left parentheses, 7, right parentheses, semicolon, 2 forward slashes, delete the last item. Line 55: System dot out dot print l n, left parentheses, double quotes, Player deleted, colon, double quotes, plus, temp, right parentheses, semicolon. Line 56: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 57: plus players dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 58: blank. Line 59: temp equals players dot delete, left parentheses, 5, right parentheses, semicolon, 2 forward slashes, delete the first item. Line 60: System dot out dot print l n, left parentheses, double quotes, Player deleted, colon, double quotes, plus temp, right parentheses, semicolon. Line 61: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 62: plus players dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus, players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 63: blank. Line 64: temp equals players dot delete, left parentheses, 7, right parentheses, semicolon, 2 forward slashes, delete from empty list. Line 65: System dot out dot print l n, left parentheses, double quotes, Player deleted, colon, double quotes, plus temp right parentheses, semicolon. Line 66: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 67: plus players dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 68: Right brace. Line 69: catch, left parentheses, Data Structure Exception, d s e 2, right parentheses. Line 70: Left brace. Line 71: System dot out dot print l n, left parentheses, d s e 2 dot get Message, Left parentheses, right parentheses, right parentheses, semicolon. Line 72: Right brace. Line 73: Right brace. Line 74: Right brace.

Back to Example

The table is titled, Methods of the Player Stack Linked List class. The table has 2 columns labeled Return Value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: push, left parentheses, Player p, right parentheses, inserts Player p at the top of the stack. Row 2. Return value: Player. Method name and argument list: pop, left parentheses, right parentheses, returns and removes the first Player of the list. If the list is empty, the method throws a Data Structure Exception. Row 3. Return value: Player. Method name and argument list: peek, left parentheses, right parentheses, returns the first Player on the list without deleting it. If the list is empty, the method throws a Data Structure Exception.

Back to Table

Line 1: Forward slash. Asterisk. The Player Stack Linked List class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Player Stack Linked List extends Shell Linked List. Line 6: Left brace. Line 7: 2 forward slashes head and number Of Items are inherited instance variables. Line 8: blank. Line 9: public Player Stack Linked List, Left parentheses, right parentheses. Line 10: Left brace. Line 11: super Left parentheses, right parentheses, semicolon. Line 12: Right brace. Line 13: blank. Line 14: Forward slash. Asterisk. Asterisk push method. Line 15: Asterisk at symbol param p Player object to insert. Line 16: Asterisk. Forward slash. Line 17: public void push left parentheses Player p right parentheses. Line 18: Left brace. Line 19: Player Node p n equals new Player Node, left parentheses p, right parentheses, semicolon. Line 20: p n dot set Next, left parentheses, head, right parentheses, semicolon. Line 21: head equals p n semicolon. Line 22: number Of Items plus plus semicolon. Line 23: Right brace. Line 24: blank. Line 25: Forward slash. Asterisk. Asterisk pop method. Line 26: Asterisk at symbol return the Player object deleted. Line 27: Asterisk. Forward slash. Line 28: public Player pop, Left parentheses, right parentheses throws Data Structure Exception. Line 29: Left brace. Line 30: if left parentheses is Empty, Left parentheses, right parentheses right parentheses. Line 31: throw new Data Structure Exception. Line 32: left parentheses double quotes, empty stack, colon, cannot be popped, double quotes right Line parentheses, semicolon. Line 33: else. Line 34: Left brace. Line 35: Player deleted equals head dot get Player, Left parentheses, right parentheses, semicolon. Line 36: head equals head dot get Next, Left parentheses, right parentheses, semicolon. Line 37: number Of Items minus minus semicolon. Line 38: return deleted, semicolon. Line 39: Right brace. Line 40: Right brace. Line 41: blank. Line 42: Forward slash. Asterisk. Asterisk peek method. Line 43: Asterisk at symbol return the Player object retrieved. Line 44: Asterisk. Forward slash. Line 45: public Player peek, Left parentheses, right parentheses throws Data Structure Exception. Line 46: Left brace. Line 47: if left parentheses is Empty, Left parentheses, right parentheses, right parentheses. Line 48: throw new Data Structure Exception. Line 49: left parentheses double quotes, empty stack, colon, cannot peek, double quotes right parentheses, semicolon. Line 50: else. Line 51: Left brace. Line 52: return head dot get Player, Left parentheses, right parentheses, semicolon. Line 53: Right brace. Line 54: Right brace. Line 55: Right brace.

Back to Example

The table is titled, Methods of the Player Queue Linked List class. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: enqueue, left parentheses, Player p, right parentheses, inserts Player p at the end of the list. Row 2. Return value: Player. Method name and argument list: dequeue, left parentheses, right parentheses, returns and removes the first player from the list. If the list is empty, the moethod throws a Data Structure Exception. Row 3. Return value: Player. Method name and argument list: peek, left parentheses, right parentheses, returns the first Player on the list, but does not delete the Player. If the list is empty, the method throws a Data Structure Exception.

Back to Table

Line 1: Forward slash. Asterisk. The Player Queue Linked List class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Player Queue Linked List extends Shell Linked List. Line

6: Left brace. Line 7: 2 forward slashes head and number Of Items are inherited instance variables. Line 8: private Player Node tail, semicolon, 2 forward slashes, last node. Line 9: blank. Line 10: public Player Queue Linked List, Left parentheses, right parentheses. Line 11: Left brace. Line 12: super, Left parentheses, right parentheses, semicolon. Line 13: tail equals null, semicolon. Line 14: Right brace. Line 15: blank. Line 16: Forward slash. Asterisk. Asterisk enqueue method. Line 17: Asterisk at symbol param p Player object to insert. Line 18: Asterisk. Forward slash. Line 19: public void enqueue, left parentheses, Player p right parentheses. Line 20: Left brace. Line 21: 2 forward slashes insert as tail. Line 22: Player Node p n equals new Player Node, left parentheses, p, right parentheses, semicolon. Line 23: if left parentheses is Empty, Left parentheses, right parentheses right parentheses. Line 24: Left brace. Line 25: tail equals p n, semicolon. Line 26: head equals p n, semicolon. Line 27: Right brace. Line 28: else. Line 29: Left brace. Line 30: tail dot set Next, left parentheses, p n, right parentheses, semicolon. Line 31: tail equals p n, semicolon. Line 32: Right brace. Line 33: number Of Items plus plus semicolon. Line 34: Right brace. Line 35: blank. Line 36: Forward slash. Asterisk. Asterisk dequeue method. Line 37: Asterisk at symbol return a reference to the Player object deleted. Line 38: Asterisk. Forward slash. Line 39: public Player dequeue, Left parentheses, right parentheses throws Data Structure Exception. Line 40: Left brace. Line 41: if left parentheses is Empty, Left parentheses, right parentheses, right parentheses. Line 42: throw new Data Structure Exception. Line 43: left parentheses, double quotes, empty queue, colon, cannot dequeue, double quotes, right parentheses, semicolon. Line 44: else. Line 45: Left brace. Line 46: Player deleted equals head dot get Player, Left parentheses, right parentheses, semicolon. Line 47: head equals head dot get Next, Left parentheses, right parentheses, semicolon. Line 48: if left parentheses minus minus number Of Items equals equals 0 right parentheses. Line 49: tail equals null, semicolon. Line 50: return deleted, semicolon. Line 51: Right brace. Line 52: Right brace. Line 53: blank. Line 54: Forward slash. Asterisk. Asterisk peek method. Line 55: Asterisk at symbol return a reference to the Player object retrieved. Line 56: Asterisk. Forward slash. Line 57: public Player peek, Left parentheses, right parentheses throws Data Structure Exception. Line 58: Left brace. Line 59: if left parentheses is Empty, Left parentheses, right parentheses, right parentheses. Line 60: throw new Data Structure Exception. Line 61: left parentheses, double quotes, empty queue, colon, cannot peek, double quotes, right parentheses, semicolon. Line 62: else. Line 63: Left brace. Line 64: return head dot get Player, Left parentheses, right parentheses, semicolon. Line 65: Right brace. Line 66: Right brace. Line 67: Right brace.

Back to Example

The table is titled, Methods of the Array Stack Class. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: boolean. Method name and argument list: push, left parentheses, Player p, right parentheses, inserts Player p at the top of the stack, if the stack is not full. Returns ture if the insertion was successful; false otherwise. Row 2. Return value: Player. Method name and argument list: pop, left parentheses, right parentheses, removes and returns the Player at the top of the stack, if the stack is not empty. If the stack is empty, the method throws a Data Structure Exception. Row 3. Return value: Player. Method name and argument list: peek, left parentheses, right parentheses, returns the Player at the top of the stack if the stack is not empty. If the stack is empty, the method throws a Data Structure Exception.

Back to Table

Line 1: Forward slash. Asterisk. The Array Stack class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Array Stack. Line 6: Left brace. Line 7: private static final int STACK underscore SIZE equals 100, semicolon. 2 forward slashes, maximum array size. Line 8: private Player left square bracket right square bracket stack, semicolon. 2 forward slashes, array of Player objects. Line 9: private int top, semicolon. 2 forward slashes last used index, semicolon top of the stack. Line 10: blank. Line 11: public Array Stack, Left parentheses, right parentheses. Line 12: Left brace. Line 13: stack equals new Player, left square bracket, STACK underscore SIZE, right square bracket, semicolon. Line 14: top equals negative 1, semicolon. 2 forward slashes, stack is empty. Line 15: Right brace. Line 16: blank. Line 17: Forward slash. Asterisk. Asterisk push method. Line 18: Asterisk at symbol param p Player object to insert. Line 19: Asterisk at symbol return true if insertion was successful false otherwise. Line 20: Asterisk. Forward slash. Line 21: public boolean push, left parentheses, Player p, right parentheses. Line 22: Left brace. Line 23: if left parentheses, exclamation mark, is Full, Left parentheses, right parentheses, right parentheses, 2 forward slashes, is there room to insert, question mark. Line 24: Left brace. Line 25: stack, left square bracket, plus plus top, right square bracket, equals p, semicolon. Line 26: return true, semicolon. Line 27: Right brace. Line 28: else. Line 29: return false, semicolon. Line 30: Right brace. Line 31: blank. Line 32: Forward slash. Asterisk. Asterisk pop method. Line 33: Asterisk at symbol return the Player deleted. Line 34: Asterisk. Forward slash. Line 35: public Player pop, Left parentheses, right parentheses, throws Data Structure Exception. Line 36: Left brace. Line 37: if left parentheses exclamation mark, is Empty, Left parentheses, right parentheses, right Line parentheses 2 forward slashes, is there an item to delete, question mark. Line 38: return stack, left square bracket, top minus minus, right square bracket, semicolon. Line 39: else. Line 40: throw new Data Structure Exception. Line 41: left parentheses, double quotes, Stack empty, colon, cannot pop, double quotes, right parentheses, semicolon. Line 42: Right brace. Line 43: blank. Line 44: Forward slash. Asterisk. Asterisk peek method. Line 45: Asterisk at symbol return the Player at the top of the stack. Line 46: Asterisk. Forward slash. Line 47: public Player peek, Left parentheses, right parentheses throws Data Structure Exception. Line 48: Left brace. Line 49: if left parentheses exclamation mark, is Empty, Left parentheses, right parentheses, right parentheses, 2 forward slashes, stack is not empty. Line 50: return stack, left square bracket, top, right square bracket, semicolon. Line 51: else. Line 52: throw new Data Structure Exception. Line 53: left parentheses double quotes, Stack empty, colon, cannot peek, double quotes, right parentheses, semicolon. Line 54: Right brace. Line 55: blank. Line 56: Forward slash. Asterisk. Asterisk, is Empty method. Line 57: Asterisk at symbol return true if stack is empty, comma false otherwise. Line 58: Asterisk. Forward slash. Line 59: public boolean is Empty, Left parentheses, right parentheses. Line 60: Left brace. Line 61: return left parentheses, top, equals equals negative 1 right parentheses, semicolon. Line 62: Right brace. Line 63: blank. Line 64: Forward slash. Asterisk. Asterisk, is Full method. Line 65: Asterisk at symbol return true if stack is full, comma, false otherwise. Line 66: Asterisk. Forward slash. Line 67: public boolean is Full, Left parentheses, right parentheses. Line 68: Left brace. Line 69: return, left parentheses, top equals equals, left parentheses, STACK underscore SIZE minus 1, right parentheses, right parentheses, semicolon. Line 70: Right brace. Line 71: blank. Line 72: Forward slash. Asterisk. Asterisk, to String method. Line 73: Asterisk at symbol return the stack elements starting at top. Line 74: Asterisk. Forward slash. Line 75: at symbol Override. Line 76: public String to String, Left parentheses, right parentheses. Line 77: Left brace. Line 78: String stack String equals double quotes, double quotes, semicolon. Line 79: for left parentheses int i equals top, semicolon i greater than or equal to 0, semicolon, i minus minus, right parentheses. Line 80: stack String plus equals, left parentheses, i plus, double quotes, colon, double quotes, plus stack, left square bracket, i, right square bracket, plus, double quotes, backslash n, double quotes, right parentheses, semicolon. Line 81: return stack String semicolon. Line 82: Right brace. Line 83: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. Asterisk push method. Line 2: Asterisk at symbol param value, value to be pushed onto the stack. Line 3: Asterisk at symbol return true if successful, comma, false if unsuccessful. Line 4: Asterisk. Forward slash. Line 5: public boolean push, left parentheses, int value right parentheses. Line 6: Left brace. Line 7: 2 forward slashes, 5 asterisks, 1. Student code starts here, 5 asterisks. Line 8: 2 forward slashes, stack is an int array instance variable representing. Line 9: 2 forward slashes, the array that stores our stack. Line 10: 2 forward slashes, top is an instance variable representing. Line 11: 2 forward slashes, the index of the top of the stack. Line 12: 2 forward slashes CAPACITY is a static constant representing. Line 13: 2 forward slashes the size of the array stack. Line 14: 2 forward slashes The push method adds the argument value. Line 15: 2 forward slashes to the top of the stack comma if it is possible. Line 16: 2 forward slashes code the push method here. Line 17: 2 forward slashes end of student code comma part 1. Line 18: Right brace. Line 19: Forward slash. Asterisk. Asterisk pop method. Line 20: Asterisk at symbol return the value of the top element of the stack comma if. Line 21: Asterisk successful. Line 22: Asterisk. Forward slash. Line 23: public int pop Left parentheses, right parentheses throws Data Structure Exception. Line 24: Left brace. Line 25: 2 forward slashes, 5 asterisks, 2. Student code restarts here, 5 asterisks. Line 26: 2 forward slashes, stack is an int array instance variable representing. Line 27: 2 forward slashes the array that stores our stack. Line 28: 2 forward slashes top is an instance variable representing. Line 29: 2 forward slashes the index of the top of the stack. Line 30: 2 forward slashes CAPACITY is a static constant representing. Line 31: 2 forward slashes the size of the array stack. Line 32: 2 forward slashes The pop method deletes the element. Line 33: 2 forward slashes at the top of the stack comma if it is possible. Line 34: 2 forward slashes code the pop method here. Line 35: 2 forward slashes end of student code comma part 2. Line 36: Right brace.

Back to Example

The table is titled, Methods of the Array Queue Class. The table has 2 columns labeled Return Value and Method name and argument list. The row entries are as follows. Row 1. Return value: boolean. Method name and argument list: enqueue, left parentheses, Player p, right parentheses, inserts Player p at the back of the queue if the queue is not full. Returns true if the insertion was successful, false otherwise. Row 2. Return value: Player. Method name and argument list: dequeue, left parentheses, right parentheses, returns and removes the Player at the front of the queue. If the queue is empty, the method throws a Data Structure Exception. Row 3. Return value: Player. Method name and argument list: peek, left parentheses, right parentheses, returns the Player at the front of the queue. If the queue is empty, the method throws a Data Structure Exception.

Back to Table

Line 1: Forward slash. Asterisk. The Array Queue class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Array Queue. Line 6: Left brace. Line 7: private static final int QUEUE underscore SIZE equals 8, semicolon. Line 8: private Player left square bracket right square bracket queue, semicolon. Line 9: private int front, semicolon. Line 10: private int back, semicolon. Line 11: private int number Of Items, semicolon. Line 12: blank. Line 13:

public Array Queue, Left parentheses, right parentheses. Line 14: Left brace. Line 15: queue equals new Player, left square bracket, QUEUE underscore SIZE, right square bracket, semicolon. Line 16: front equals 0, semicolon. Line 17: back equals QUEUE underscore SIZE minus 1semicolon. Line 18: number Of Items equals 0, semicolon. Line 19: Right brace. Line 20: blank. Line 21: public boolean is Full, Left parentheses, right parentheses. Line 22: Left brace. Line 23: return left parentheses number Of Items equals equals QUEUE underscore SIZE, right parentheses, semicolon. Line 24: Right brace. Line 25: blank. Line 26: public boolean is Empty, Left parentheses, right parentheses. Line 27: Left brace. Line 28: return left parentheses number Of Items equals equals 0, right parentheses, semicolon. Line 29: Right brace. Line 30: blank. Line 31: Forward slash. Asterisk. Asterisk enqueue method. Line 32: Asterisk at symbol param p the Player to insert. Line 33: Asterisk at symbol return true if list is not full, comma, false otherwise. Line 34: Asterisk. Forward slash. Line 35: public boolean enqueue left parentheses Player p right parentheses. Line 36: Left brace. Line 37: if left parentheses exclamation mark, is Full, Left parentheses, right parentheses right parentheses. Line 38: Left brace. Line 39: queue, left square bracket, left parentheses, back plus 1 right parentheses percentage sign QUEUE underscore SIZE, right square bracket, equals p, semicolon. Line 40: back equals left parentheses back plus 1 right parentheses percentage sign QUEUE underscore SIZE, semicolon. Line 41: number Of Items plus plus semicolon. Line 42: return true, semicolon. Line 43: Right brace. Line 44: else. Line 45: return false, semicolon. Line 46: Right brace. Line 47: blank. Line 48: Forward slash. Asterisk. Asterisk dequeue method. Line 49: Asterisk at symbol return the Player deleted. Line 50: Asterisk. Forward slash. Line 51: public Player dequeue, Left parentheses, right parentheses throws Data Structure Exception. Line 52: Left brace. Line 53: if left parentheses exclamation mark, is Empty, Left parentheses, right parentheses right parentheses. Line 54: Left brace. Line 55: front equals left parentheses front plus 1 right parentheses percentage sign QUEUE underscore SIZE, semicolon. Line 56: number Of Items minus minus semicolon. Line 57: return queue, left square bracket, left parentheses, QUEUE underscore SIZE plus front minus 1 right parentheses percentage sign QUEUE underscore SIZE right square bracket, semicolon. Line 58: Right brace. Line 59: else. Line 60: throw new Data Structure Exception. Line 61: left parentheses double quotes Queue empty, colon, cannot dequeue double quotes right parentheses, semicolon. Line 62: Right brace. Line 63: blank. Line 64: Forward slash. Asterisk. Asterisk to String method. Line 65: Asterisk at symbol return a front-to-back String representation of the queue. Line 66: Asterisk. Forward slash. Line 67: at symbol Override. Line 68: public String to String Left parentheses, right parentheses. Line 69: Left brace. Line 70: String queue String equals double quotes double quotes semicolon.

Line 71: for left parentheses int i equals front semicolon i less than front plus number Of Items semicolon i plus plus right parentheses. Line 72: queue String plus equals queue left square bracket i percentage sign QUEUE underscore SIZE right square bracket dot to String, Left parentheses, right parentheses plus double quotes backslash n double quotes semicolon. Line 73: return queue String semicolon. Line 74: Right brace. Line 75: Right brace.

Back to Example

The table has 2 columns labeled Array and Linked List. The row entries are as follows. Row 1. Easily expanded. Array: No. Linked List: Yes. Row 2. Direct access to every item. Array: Yes. Linked List: No. Row 3. Easy to code. Array: Yes. Linked List: No.

Back to Table

Dot, dot, dot. Line 105: blank. Line 106: Forward slash. Asterisk. Asterisk compare To method. Line 107: Asterisk at symbol param p a Player to compare to this Player. Line 108: Asterisk at symbol return an int comma minus 1 comma 0 comma or 1 as this Player is less than comma. Line 109: Asterisk equal to comma or greater than player. Line 110: Asterisk. Forward slash. Line 111: public int compare To, left parentheses Player p right parentheses. Line 112: Left brace. Line 113: if left parentheses i d equals equals p dot i d 2 ampersands name dot equals left parentheses p dot name right parentheses. Line 114: 2 ampersands game dot equals left parentheses p dot game right parentheses right parentheses. Line 115: return 0 semicolon. Line 116: else if left parentheses i d greater than p dot i d, 2 pipes. Line 117: left parentheses i d equals equals p dot i d 2 ampersands name dot compare To left parentheses p dot name right parentheses greater than 0 right parentheses 2 pipes. Line 118: left parentheses i d equals equals p dot i d 2 ampersands name dot equals left parentheses p dot name right parentheses. Line 119: 2 ampersands game dot compare To left parentheses p dot game right parentheses greater than 0 right parentheses right parentheses. Line 120: return 1 semicolon. Line 121: else. Line 122: return minus 1 semicolon. Line 123: Right brace. Line 124: Right brace.

Back to Example

The table is titled, Methods of the Player Sorted Linked List class. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: insert, left parentheses, Player p, right parentheses, inserts Player p in a location that keeps the list sorted in ascending order. Row 2. Return value: Player. Method name and argument list: delete, left parentheses, int search I D, right parentheses, returns and removes the first Player of the list with an i d equal to search I D. If there is no such Player on the list, the method throws a Data Structure Exception.

Back to Table

Line 1: Forward slash. Asterisk. The Player Sorted Linked List class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Player Sorted Linked List extends Shell Linked List. Line 6: Left brace. Line 7: 2 forward slashes head and number Of Items are inherited instance variables. Line 8: blank. Line 9: public Player Sorted Linked List Left parentheses, right parentheses. Line 10: Left brace. Line 11: super Left parentheses, right parentheses semicolon. Line 12: Right brace. Line 13: blank. Line 14: Forward slash. Asterisk. Asterisk insert method. Line 15: Asterisk at symbol param p Player object to insert. Line 16: Asterisk. Forward slash. Line 17: public void insert left parentheses Player p right parentheses. Line 18: Left brace. Line 19: Player Node p n equals new Player Node left parentheses p right parentheses semicolon. Line 20: blank. Line 21: 2 forward slashes we will insert after previous and before current. Line 22: Player Node current equals head semicolon. Line 23: Player Node previous equals null semicolon. Line 24: while left parentheses current exclamation mark, equals null. Line 25: 2 ampersands current dot get Player, Left parentheses, right parentheses dot compare To, left parentheses p right parentheses less than 0 right parentheses. Line 26: Left brace. Line 27: previous equals current semicolon. Line 28: current equals current dot get Next, Left parentheses, right parentheses, semicolon.

Line 29: Right brace. Line 30: blank. Line 31: if left parentheses previous equals equals null right parentheses 2 forward slashes insert as head. Line 32: Left brace. Line 33: p n dot set Next left parentheses head right parentheses, semicolon. Line 34: head equals p n semicolon. Line 35: Right brace. Line 36: else. Line 37: Left brace. Line 38: p n dot set Next left parentheses current right parentheses semicolon. Line 39: previous dot set Next, left parentheses, p n, right parentheses, semicolon. Line 40: Right brace. Line 41: number Of Items plus plus semicolon. Line 42: Right brace. Line 43: blank. Line 44: Forward slash. Asterisk. Asterisk delete method. Line 45: Asterisk at symbol param search I D i d of Player to delete. Line 46: Asterisk at symbol return the Player deleted. Line 47: Asterisk. Forward slash. Line 48: public Player delete left parentheses int search I D right parentheses. Line 49: throws Data Structure Exception. Line 50: Left brace. Line 51: Player Node current equals head semicolon. Line 52: Player Node previous equals null semicolon. Line 53: while left parentheses current exclamation mark, equals null. Line 54: 2 ampersands current dot get Player, Left parentheses, right parentheses dot get I D, Left parentheses, right Line parentheses exclamation mark, equals search I D right parentheses. Line 55: Left brace. Line 56: if left parentheses current dot get Player, Left parentheses, right parentheses dot get I D, Left parentheses, right parentheses, greater than, search I D, right parentheses. Line 57: throw new Data Structure Exception. Line 58: left parentheses search I D plus double quotes not found, colon, cannot be deleted, double quotes, right parentheses, semicolon. Line 59: previous equals current, semicolon. Line 60: current equals current dot get Next, Left parentheses, right parentheses, semicolon. Line 61: Right brace. Line 62: blank. Line 63: if left parentheses current equals equals null right parentheses 2 forward slashes not found. Line 64: throw new Data Structure Exception. Line 65: left parentheses search I D plus double quotes not found, colon, cannot be deleted, double quotes, right parentheses, semicolon. Line 66: else 2 forward slashes search I D found at Player at node current. Line 67: Left brace. Line 68: if left parentheses current equals equals head right parentheses. Line 69: head equals head dot get Next, Left parentheses, right parentheses, semicolon, 2 forward slashes delete head. Line 70: else. Line 71: previous dot set Next, left parentheses, current dot get Next, Left parentheses, right parentheses right parentheses, semicolon. Line 72: blank. Line 73: number Of Items minus minus semicolon. Line 74: return current dot get Player, Left parentheses, right parentheses, semicolon. Line 75: Right brace. Line 76: Right brace. Line 77: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. The Player Sorted Linked List Test class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Player Sorted Linked List Test. Line 6: Left brace.

Line 7: public static void main, left parentheses String, left square bracket, right square bracket, a r g s right parentheses. Line 8: Left brace. Line 9: Player p 1 equals new Player, left parentheses, 7, comma, double quotes, Sarah, double quotes, comma, double quotes, Mario, double quotes, right parentheses, semicolon. Line 10: Player p 2 equals new Player, left parentheses, 2, comma, double quotes, Jin, double quotes, comma, double quotes, Golf, double quotes, right parentheses, semicolon. Line 11: Player p 3 equals new Player, left parentheses, 5, comma, double quotes, Ajay, double quotes, comma, double quotes, Sonic, double quotes, right parentheses, semicolon. Line 12: Player p 4 equals new Player, left parentheses, 8, comma, double quotes, Gino, double quotes, comma, double quotes, Diablo, double quotes, right parentheses, semicolon. Line 13: blank. Line 14: 2 forward slashes, construct empty Player Sorted Linked List. Line 15: Player Sorted Linked List players equals. Line 16: new Player Sorted Linked List, Left parentheses, right parentheses, semicolon. Line 17: blank. Line 18: System dot out dot print l n left parentheses double quotes Number of items in the list, colon, double quotes. Line 19: plus players dot get Number Of Items, Left parentheses, right parentheses, plus double quotes, backslash n, double quotes plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 20: blank. Line 21: System dot out dot print l n left parentheses double quotes inserting double quotes plus p 1 right parentheses, semicolon. Line 22: players dot insert, left parentheses, p 1 right parentheses, semicolon, 2 forward slashes, insert in empty list. Line 23: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 24: plus players dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 25: blank. Line 26: System dot out dot print l n, left parentheses, double quotes, inserting double quotes, plus p 2, right parentheses, semicolon. Line 27: players dot insert, left parentheses, p 2, right parentheses, semicolon, 2 forward slashes, insert at the beginning of the list. Line 28: System dot out dot print l n, left parentheses double quotes, Number of items in the list, colon, double quotes. Line 29: plus players dot get Number Of Items, Left parentheses, right parentheses, plus double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 30: blank. Line 31: System dot out dot print l n, left parentheses, double quotes, inserting double quotes plus p 3 right parentheses, semicolon. Line 32: players dot insert, left parentheses p 3 right parentheses, semicolon, 2 forward slashes insert in the middle of the list. Line 33: System dot out dot print l n left parentheses double quotes, Number of items in the list, colon, double quotes. Line 34: plus players dot get Number Of Items, Left parentheses, right parentheses, plus, double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 35: blank. Line 36: System dot out dot print l n left parentheses double quotes, inserting double quotes plus p 4 right parentheses, semicolon. Line 37: players dot insert, left parentheses, p 4, right parentheses, semicolon, 2 forward slashes, insert at the end of the list. Line 38: System dot out dot print l n left parentheses double quotes Number of items in the list, colon, double quotes. Line 39: plus players dot get Number Of Items, Left parentheses, right parentheses plus double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses right parentheses, semicolon. Line 40: Right brace. Line 41: Right brace.

Back to Example

Line 1: public void insert left parentheses int value right parentheses. Line 2: Left brace. Line 3: 2 forward slashes, 5 asterisks, Student writes the body of this method, 5 asterisks. Line 4: 2 forward slashes code the insert method of a linked list of ints. Line 5: 2 forward slashes the int to insert in the linked list is value. Line 6: 2 forward slashes. Line 7: 2 forward slashes Student code starts here. Line 8: 2 forward slashes. Line 9: 2 forward slashes. Line 10: 2 forward slashes End of student code comma part 1. Line 11: 2 forward slashes. Line 12: Right brace. Line 13: public boolean delete, left parentheses, int value, right parentheses. Line 14: Left brace. Line 15: 2 forward slashes 5 asterisks Student writes the body of this method 5 asterisks. Line 16: 2 forward slashes code the delete method of a linked list of ints. Line 17: 2 forward slashes the int to delete in the linked list is value. Line 18: 2 forward slashes if deletion is successful, comma, return true. Line 19: 2 forward slashes, otherwise comma return false. Line 20: 2 forward slashes. Line 21: 2 forward slashes Student code starts here. Line 22: 2 forward slashes. Line 23: return true, semicolon, 2 forward slashes, replace this return statement. Line 24: 2 forward slashes. Line 25: 2 forward slashes End of student code, comma, part 2. Line 26: 2 forward slashes. Line 27: Right brace.

Back to Example

A table titled, Constructors and Methods of the Node Class. The first table summmarizes the Constructors of the Node Class. The table has 2 columns labeled Class and Constructor and argument list. The row entries are as follows. Row 1. Class: Node. Constructor and argument list: Node, left parentheses, int data, right parentheses, constructs a new Node object whose data instance variable is data. The pointer to the next node is set to the value null. Row 2. Class: Node. Constructor and argument list: Node, left parentheses, int data, comma, Node next Node, right parentheses, constructs a new Node object whose data instance variable is data. The Node points to next Node. The second table summarizes the methods of the Node class. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: Node. Method name and argument list: set Next, left parentheses, Node next Node, right parentheses, sets the Node object reference pointed to by this Node to next Node. Row 2. Return value: Node. Method name and argument list: set Data, left parentheses, int data, right parentheses, sets the data instance variable to data. Row 3. Return value: Node. Method name and argument list: get Next, left parentheses, right parentheses, returns an object reference to the Node pointed to by this Node. Row 4. Return value: int. Method name and argument list: get Data, left parentheses, right parentheses, returns the data stored in this Node.

Back to Table

Program code. The code contains the following keywords: public, class, private. Line 1: Forward slash, asterisk. The Player Node class. Line 2, indented once: Anderson, comma, Franceschi. Line 3: Asterisk, Forward slash. Line 4: blank. Line 5: public class Player Node. Line 6: left brace. Line 7, indented once: private Player player, semicolon. 2 forward slashes, the player at that node. Line 8, indented once: private Player Node next, semicolon. 2 forward slashes, the next Player Node. Line 9, indented once: private Player Node previous, semicolon. 2 forward slashes, the previous Player Node. Line 10: blank. Line 11, indented once: 2 forward slashes, constructors. Line 12, indented once: 2 forward slashes, accessors are get Player, comma, get Next, comma, get Previous. Line 13, indented once: 2 forward slashes, mutators are set Player, comma, set Next, comma, set Previous. Line 14: right brace.

Back to Example

Line 1: Forward slash. Asterisk. Asterisk The Node class. Line 2: Asterisk Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Node left angled bracket T right angled bracket. Line 6: Left brace. Line 7: private T data, semicolon. Line 8: private Node, left angled bracket, T, right angled bracket, next, semicolon. Line 9: blank. Line 10: Forward slash. Asterisk. Asterisk default constructor. Line 11: Asterisk sets data and next to null. Line 12: Asterisk. Forward slash. Line 13: public Node, Left parentheses, right parentheses. Line 14: Left brace. Line 15: data equals null, semicolon. Line 16: next equals null, semicolon. Line 17: Right brace. Line 18: blank. Line 19: Forward slash. Asterisk. Asterisk constructor. Line 20: Asterisk at symbol param data reference to data. Line 21: Asterisk sets next to null. Line 22: Asterisk. Forward slash. Line 23: public Node, left parentheses, T data, right parentheses. Line 24: Left brace. Line 25: set Data, left parentheses, data, right parentheses, semicolon. Line 26: next equals null, semicolon. Line 27: Right brace. Line 28: blank. Line 29: Forward slash. Asterisk. Asterisk accessor for data. Line 30: Asterisk at symbol return reference to data item. Line 31: Asterisk. Forward slash. Line 32: public T get Data, Left parentheses, right parentheses. Line 33: Left brace. Line 34: return data, semicolon. Line 35: Right brace. Line 36: blank. Line 37: Forward slash. Asterisk. Asterisk accessor for next. Line 38: Asterisk at symbol return next. Line 39: Asterisk. Forward slash. Line 40: public Node, left angled bracket, T, right angled bracket, get Next, Left parentheses, right parentheses. Line 41: Left brace. Line 42: return next, semicolon. Line 43: Right brace. Line 44: blank. Line 45: Forward slash. Asterisk. Asterisk mutator for data. Line 46: Asterisk at symbol param data reference to new data item. Line 47: Asterisk at symbol return reference to this object. Line 48: Asterisk. Forward slash. Line 49: public Node set Data, left parentheses, T data, right parentheses. Line 50: Left brace. Line 51: this dot data equals data, semicolon. Line 52: return next, semicolon. Line 53: Right brace. Line 54: blank. Line 55: Forward slash. Asterisk. Asterisk mutator for next. Line 56: Asterisk at symbol param reference to next Node. Line 57: Asterisk at symbol return reference to this object. Line 58: Asterisk. Forward slash. Line 59: public Node set Next, left parentheses, Node, left angled bracket, T, right angled bracket next right parentheses. Line 60: Left brace. Line 61: this dot next equals next, semicolon. Line 62: return this, semicolon. Line 63: Right brace. Line 64: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. The Shell Linked List class. Line 2: Anderson comma Franceschi. Line 3:

Asterisk. Forward slash. Line 4: blank. Line 5: public abstract class Shell Linked List, left angled bracket, T, right angled bracket. Line 6: Left brace. Line 7: protected Node, left angled bracket, T, right angled bracket, head, semicolon. Line 8: protected int number Of Items semicolon. Line 9: blank. Line 10: Forward slash. Asterisk. Asterisk constructor. Line 11: Asterisk sets head to null and number Of Items to 0. Line 12: Asterisk. Forward slash. Line 13: public Shell Linked List, Left parentheses, right parentheses. Line 14: Left brace. Line 15: head equals null, semicolon. Line 16: number Of Items equals 0 semicolon. Line 17: Right brace. Line 18: blank. Line 19: Forward slash. Asterisk. Asterisk accessor for number Of Items. Line 20: Asterisk at symbol return number Of Items. Line 21: Asterisk. Forward slash. Line 22: public int get Number Of Items, Left parentheses, right parentheses. Line 23: Left brace. Line 24: return number Of Items semicolon. Line 25: Right brace. Line 26: blank. Line 27: Forward slash. Asterisk. Asterisk is Empty method. Line 28: Asterisk at symbol return true if no items in list, semicolon false otherwise. Line 29: Asterisk. Forward slash. Line 30: public boolean is Empty, Left parentheses, right parentheses. Line 31: Left brace. Line 32: return left parentheses number Of Items equals equals 0 right parentheses semicolon. Line 33: Right brace. Line 34: blank. Line 35: Forward slash. Asterisk. Asterisk to String method. Line 36: Asterisk at symbol return all items in the list. Line 37: Asterisk. Forward slash. Line 38: at symbol Override. Line 39: public String to String Left parentheses, right parentheses. Line 40: Left brace. Line 41: String list String equals double quotes double quotes semicolon. Line 42: Node left angled bracket T right angled bracket current equals head semicolon. Line 43: for left parentheses int i equals 0 semicolon i left angled bracket number Of Items semicolon i plus plus right parentheses. Line 44: Left brace. Line 45: list String plus equals current dot get Data Left parentheses, right parentheses dot to String Left parentheses, right parentheses plus double quotes backslash n double quotes semicolon. Line 46: current equals current dot get Next, Left parentheses, right parentheses, semicolon. Line 47: Right brace. Line 48: return list String semicolon. Line 49: Right brace. Line 50: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. The Generic Linked List class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Generic Linked List left angled bracket T right angled bracket extends Shell Linked List left angled bracket T right angled bracket. Line 6: Left brace. Line 7: 2 forward slashes head and number Of Items are inherited instance variables. Line 8: blank. Line 9: Forward slash. Asterisk. Asterisk constructor. Line 10: Asterisk calls constructor of Shell Linked List. Line 11: Asterisk. Forward slash. Line 12: public Generic Linked List Left parentheses, right parentheses. Line 13: Left brace. Line 14: super Left parentheses, right parentheses semicolon. Line 15: Right brace. Line 16: blank. Line 17: Forward slash. Asterisk. Asterisk insert method. Line 18: Asterisk at symbol param item T object to insert. Line 19: Asterisk. Forward slash. Line 20: public void insert, left parentheses T item right parentheses. Line 21: Left brace. Line 22: 2 forward slashes insert as head. Line 23: Node, left angled bracket, T, right angled bracket n d equals new Node, left angled bracket, T, right angled bracket, left parentheses, item right parentheses, semicolon. Line 24: n d dot set Next, left parentheses, head, right parentheses, semicolon. Line 25: head equals n d, semicolon. Line 26: number Of Items plus plus semicolon. Line 27: Right brace. Line 28: blank. Line 29: Forward slash. Asterisk. Asterisk delete method. Line 30: Asterisk at symbol param item T object to delete. Line 31: Asterisk at symbol return true if the deletion was successful comma false otherwise. Line 32: Asterisk. Forward slash. Line 33: public boolean delete, left parentheses T item right parentheses. Line 34: Left brace. Line 35: Node, left angled bracket, T, right angled bracket, current equals head, semicolon. Line 36: Node, left angled bracket, T, right angled bracket, previous equals null, semicolon. Line 37: while left parentheses current exclamation mark, equals null. Line 38: 2 ampersands exclamation mark, left parentheses item dot equals left parentheses current dot get Data, Left parentheses, right parentheses, right parentheses, right parentheses, right parentheses. Line 39: Left brace. Line 40: previous equals current, semicolon. Line 41: current equals current dot get

Next, Left parentheses, right parentheses, semicolon. Line 42: Right brace. Line 43: blank. Line 44: if left parentheses current equals equals null right parentheses 2 forward slashes not found. Line 45: return false semicolon. Line 46: else. Line 47: Left brace. Line 48: if left parentheses current equals equals head right parentheses. Line 49: head equals head dot get Next, Left parentheses, right parentheses, semicolon 2 forward slashes delete head. Line 50: else. Line 51: previous dot set Next, left parentheses current dot get Next Left parentheses, right parentheses, right parentheses, semicolon. Line 52: blank. Line 53: number Of Items minus minus semicolon. Line 54: return true, semicolon. Line 55: Right brace. Line 56: Right brace. Line 57: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. The Linked List Test class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Linked List Test. Line 6: Left brace. Line 7: public static void main, left parentheses, String left square bracket right square bracket a r g s right parentheses. Line 8: Left brace. Line 9: Player p 1 equals new Player, left parentheses, 7, comma, double quotes, Sarah, double quotes, comma, double quotes, Mario, double quotes, right parentheses, semicolon. Line 10: Player p 2 equals new Player, left parentheses, 2, comma, double quotes, Jin, double quotes, comma, double quotes, Golf, double quotes right parentheses, semicolon. Line 11: Player p 3 equals new Player, left parentheses, 5, comma, double quotes, Ajay, double quotes, comma, double quotes, Sonic, double quotes, right parentheses, semicolon. Line 12: blank. Line 13: 2 forward slashes construct empty Linked List of Player objects. Line 14: Generic Linked List, left angled bracket, Player, right angled bracket, players equals new Generic Linked List, left angled bracket, Player, right angled bracket, Left parentheses, right parentheses, semicolon. Line 15: System dot out dot print l n left parentheses double quotes Number of items in the list, colon, double quotes. Line 16: plus players dot get Number Of Items Left parentheses, right parentheses plus double quotes backslash n double quotes plus players dot to String, Left parentheses, right parentheses right parentheses semicolon. Line 17: blank. Line 18: players dot insert left parentheses p 1 right parentheses semicolon 2 forward slashes insert in empty list. Line 19: System dot out dot print l n left parentheses double quotes Number of items in the list: double quotes. Line 20: plus players dot get Number Of Items Left parentheses, right parentheses plus double quotes backslash n double quotes plus players dot to String Left parentheses, right parentheses right parentheses semicolon. Line 21: blank. Line 22: players dot insert left parentheses p 2 right parentheses semicolon 2 forward slashes insert in list of one item. Line 23: System dot out dot print l n left parentheses double quotes Number of items in the list, colon, double quotes. Line 24: plus players dot get Number Of Items, Left parentheses, right parentheses, plus double quotes, backslash n, double quotes, plus, players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 25: blank. Line 26: players dot insert, left parentheses, p 3, right parentheses, semicolon, 2 forward slashes, insert in list of two items. Line 27: System dot out dot print l n left parentheses double quotes Number of items in the list, colon, double quotes. Line 28: plus players dot get Number Of Items, Left parentheses, right parentheses, plus double quotes backslash n double quotes plus players dot to String, Left parentheses, right parentheses right parentheses, semicolon. Line 29: blank. Line 30: if left parentheses players dot delete, left parentheses, p 2 right parentheses right parentheses 2 forward slashes delete in the middle. Line 31: System dot out dot print l n left parentheses double quotes Player successfully deleted, colon, double quotes right parentheses, semicolon. Line 32: System dot out dot print l n left parentheses double quotes Number of items in the list, colon, double quotes. Line 33: plus players dot get Number Of Items, Left parentheses, right parentheses, plus double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 34: blank. Line 35: if left parentheses players dot delete left parentheses p 3 right parentheses right parentheses 2 forward slashes delete at the beginning. Line 36: System dot out dot print l n left parentheses double quotes player successfully deleted, colon, double quotes right parentheses, semicolon. Line 37: System dot out dot print l n, left parentheses, double quotes, Number of items in the list, colon, double quotes. Line 38: plus players dot get Number Of Items, Left parentheses, right parentheses, plus double quotes, backslash n, double quotes, plus players dot to String, Left parentheses, right parentheses, right parentheses, semicolon. Line 39: Right brace. Line 40: Right brace.

Back to Example

The table is titled, Methods of the Recursive Linked List Class. The table has 2 columns labeled Return Value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: insert, left parentheses, T item, right parentheses, inserts item at the beginning of the list. Row 2. Return value: boolean. Method name and argument list: delete, left parentheses, T item, right parentheses, removes the first object of the list that matches item and returns true. If there is no such object in the list, the method returns false.

Back to Table

Line 1: Forward slash. Asterisk. The Recursive Linked List class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Recursive Linked List, left angled bracket, T, right angled bracket. Line 6: Left brace. Line 7: private T first, semicolon. Line 8: private Recursive Linked List, left angled bracket, T, right angled bracket, rest semicolon. Line 9: blank. Line 10: public Recursive Linked List, Left parentheses, right parentheses. Line 11: Left brace. Line 12: first equals null, semicolon. Line 13: rest equals null, semicolon. Line 14: Right brace. Line 15: blank. Line 16: Forward slash. Asterisk. Asterisk insert method. Line 17: Asterisk at symbol param item object to insert at beginning of list. Line 18: Asterisk. Forward slash. Line 19: public void insert, left parentheses, T, item right parentheses. Line 20: Left brace. Line 21: if left parentheses is Empty Left parentheses, right parentheses right parentheses 2 forward slashes is list empty, question mark. Line 22: first equals item semicolon. Line 23: else. Line 24: Left brace. Line 25: Recursive Linked List left angled bracket T right angled bracket temp List equals. Line 26: new Recursive Linked List, left angled bracket, T, right angled bracket, Left parentheses, right parentheses, semicolon. Line 27: temp List dot first equals first, semicolon. Line 28: temp List dot rest equals rest, semicolon. Line 29: first equals item, semicolon. Line 30: rest equals temp List, semicolon. Line 31: Right brace. Line 32: Right brace. Line 33: blank. Line 34: Forward slash. Asterisk. Asterisk delete method. Line 35: Asterisk at symbol param item the T object to delete. Line 36: Asterisk at symbol return true if item is deleted comma false otherwise. Line 37: Asterisk. Forward slash. Line 38: public boolean delete, left parentheses T item right parentheses. Line 39: Left brace. Line 40: if left parentheses is Empty Left parentheses, right parentheses right parentheses 2 forward slashes is list empty, question mark. Line 41: return false, semicolon. Line 42: else if left parentheses first dot equals, left parentheses, item right parentheses right parentheses 2 forward slashes found it. Line 43: Left brace. Line 44: T temp equals first semicolon. Line 45: if left parentheses rest equals equals null right parentheses. Line 46: first equals null semicolon. Line 47: else 2 forward slashes rest not null. Line 48: Left brace. Line 49: first equals rest dot first, semicolon. Line 50: rest equals rest dot rest, semicolon. Line 51: Right brace. Line 52: return true semicolon. Line 53: Right brace. Line 54: else if left parentheses rest equals equals null right parentheses. Line 55: return false semicolon. Line 56: else 2 forward slashes try to delete in rest. Line 57: return rest dot delete left parentheses item right parentheses semicolon. Line 58: Right brace. Line 59: blank. Line 60: Forward slash. Asterisk. Asterisk is Empty method. Line 61: Asterisk at symbol return true if the list has no elements, semicolon. Line 62: Asterisk false, comma otherwise. Line 63: Asterisk. Forward slash. Line 64: public boolean is Empty Left parentheses, right parentheses. Line 65: Left brace. Line 66: return left parentheses first equals equals null right parentheses semicolon. Line 67: Right brace. Line 68: blank. Line 69: Forward slash. Asterisk. Asterisk to String method. Line 70: Asterisk at symbol return a String listing the elements in the list. Line 71: Asterisk. Forward slash. Line 72: at symbol Override. Line 73: public String to String, Left parentheses, right parentheses. Line 74: Left brace. Line 75: String list String equals double quotes double quotes semicolon. Line 76: if left parentheses first exclamation mark, equals null right parentheses. Line 77: Left brace. Line 78: list String equals first dot to String, Left parentheses, right parentheses plus double quotes, backslash n, double quotes, semicolon. Line 79: if left parentheses rest exclamation mark, equals null right parentheses. Line 80: list String plus equals rest dot to String, Left parentheses, right parentheses, semicolon. Line 81: Right brace. Line 82: return list String semicolon. Line 83: Right brace. Line 84: Right brace.

Back to Example

Line 1: Forward slash. Asterisk. The Recursive Linked List Test class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Recursive Linked List Test. Line 6: Left brace. Line 7: public static void main, left parentheses, String left square bracket right square bracket a r g s right parentheses. Line 8: Left brace. Line 9: Player p 1 equals new Player, left parentheses, 7, comma, double quotes, Sarah, double quotes, comma, double quotes, Mario, double quotes, right parentheses, semicolon. Line 10: Player p 2 equals new Player, left parentheses, 2, comma, double quotes, Jin, double quotes, comma, double quotes, Golf, double quotes, right parentheses, semicolon. Line 11: Player p 3 equals new Player, left parentheses, 5, comma, double quotes, Ajay, double quotes, comma, double quotes, Sonic, double quotes, right parentheses, semicolon. Line 12: blank. Line 13: Recursive Linked List, left angled bracket, Player, right angled bracket, players equals. Line 14: new Recursive Linked List left angled bracket Player right angled bracket Left parentheses, right parentheses semicolon. Line 15: System dot out dot print l n left parentheses double quotes The list is backslash n double quotes. Line 16: plus left parentheses players dot is Empty, Left parentheses, right parentheses, question mark, double quotes, empty, backslash n, double quotes, colon, player dot to String, Left parentheses, right parentheses right parentheses right parentheses semicolon. Line 17: blank. Line 18: players dot insert left parentheses p 1 right parentheses semicolon. Line 19: System dot out dot print l n left parentheses double quotes Inserting double quotes plus p 1 right parentheses semicolon. Line 20: System dot out dot print l n left parentheses double quotes The list is backslash n double quotes. Line 21: plus left parentheses players dot is Empty, Left parentheses, right parentheses, question mark, double quotes, empty, backslash n, double quotes, colon, players dot to String, Left parentheses, right parentheses, right parentheses, right parentheses, semicolon. Line 22: blank. Line 23: players dot insert, left parentheses, p 2, right parentheses, semicolon. Line 24: System dot out dot print l n left parentheses double quotes Inserting double quotes plus p 2 right parentheses, semicolon. Line 25: System dot out dot print l n left parentheses double quotes The list is backslash n double quotes. Line 26: plus left parentheses players dot is Empty Left parentheses, right parentheses, question mark, double quotes, empty backslash n, double quotes, colon, players dot to String Left parentheses, right parentheses right parentheses right parentheses semicolon. Line 27: blank. Line 28: players dot insert, left parentheses, p 3, right parentheses semicolon. Line 29: System dot out dot print l n left parentheses double quotes Inserting double quotes plus p 3 right parentheses semicolon. Line 30: System dot out dot print l n left parentheses double quotes The list is backslash n double quotes. Line 31: plus left parentheses players dot is Empty Left parentheses, right parentheses, question mark, double quotes empty backslash n double quotes, colon, players dot to String Left parentheses, right parentheses right parentheses right parentheses semicolon. Line 32: blank. Line 33: if left parentheses players dot delete left parentheses p 2 right parentheses right parentheses 2 forward slashes delete in middle of list. Line 34: System dot out dot print l n left parentheses double quotes Player deleted, colon, double quotes plus p 2 right parentheses semicolon. Line 35: System dot out dot print l n left parentheses double quotes The list is backslash n double quotes. Line 36: plus left parentheses players dot is Empty, Left parentheses, right parentheses, question mark, double quotes emptybackslash n double quotes, colon, players dot to String Left parentheses, right parentheses right parentheses, right parentheses semicolon. Line 37: blank. Line 38: if left parentheses players dot delete left parentheses p 1 right parentheses right parentheses 2 forward slashes delete at end of the list. Line 39: System dot out dot print l n left parentheses double quotes Player deleted, colon, double quotes plus p 1 right parentheses semicolon. Line 40: System dot out dot print l n left parentheses double quotes The list is backslash n double quotes. Line 41: plus left parentheses players dot is Empty, Left parentheses, right parentheses, question mark, double quotes, empty backslash n, double quotes, colon, players dot to String, Left parentheses, right parentheses, right parentheses, right parentheses, semicolon. Line 42: blank. Line 43: if left parentheses players dot delete left parentheses p 1 right parentheses right parentheses 2 forward slashes attempt to delete will fail. Line 44: System dot out dot print l n left parentheses double quotes The list deleted, colon, double quotes plus p 1 right parentheses semicolon. Line 45: System dot out dot print l n left parentheses double quotes, The list is backslash n, double quotes. Line 46: plus left parentheses players dot is Empty, Left parentheses, right parentheses, question mark, double quotes, empty backslash n, double quotes, colon, players dot to String, Left parentheses, right parentheses, right parentheses, right parentheses, semicolon. Line 47: blank. Line 48: if left parentheses players dot delete left parentheses p 3 right parentheses right parentheses 2 forward slashes delete only Player in list. Line 49: System dot out dot print l n left parentheses double quotes Player deleted, colon, double quotes plus p

3 right parentheses semicolon. Line 50: System dot out dot print l n left parentheses double quotes The list is backslash n double quotes. Line 51: plus left parentheses players dot is Empty Left parentheses, right parentheses, question mark, double quotes empty backslash n double quotes, colon, players dot to String Left parentheses, right parentheses right parentheses right parentheses semicolon. Line 52: blank. Line 53: if left parentheses players dot delete left parentheses p 3 right parentheses right parentheses 2 forward slashes try to delete from empty list. Line 54: System dot out dot print l n left parentheses double quotes Player deleted, colon, double quotes plus p 3 right parentheses, semicolon. Line 55: System dot out dot print l n left parentheses double quotes The list is backslash n double quotes. Line 56: plus left parentheses players dot is Empty Left parentheses, right parentheses, question mark, double quotes empty backslash n double quotes, colon, players dot to String Left parentheses, right parentheses right parentheses right parentheses semicolon. Line 57: Right brace. Line 58: Right brace.

Back to Example

The table has 2 columns labelled Order of Magnitude and Number of Statements Executed. The column labelled Number of Statements Executed is subdivided into 4 columns which are labelled n equals 10, n equals 20, n equals 1000 and n equals 1 million. The row entries are as follows. Row 1. Order of Magnitude: log n. Number of

Statements Executed, n equals 10: 2.23. Number of Statements Executed, n equals 20: 3.23. Number of Statements Executed, n equals 1000: Approximately 10. Number of Statements Executed, n equals 1 million: Approximately 20. Row 2. Order of Magnitude: n. Number of Statements Executed, n equals 10: 10. Number of Statements Executed, n equals 20: 20. Number of Statements Executed, n equals 1000: 1000. Number of Statements Executed, n equals 1 million: 10 to the power of 6. Row 3. Order of Magnitude: n log n. Number of Statements Executed, n equals 10: 22.3. Number of Statements Executed, n equals 20: 64.6. Number of Statements Executed, n equals 1000: Approximately 10000. Number of Statements Executed, n equals 1 million: Approximately 20 times 10 to the power of 6. Row 4. Order of Magnitude: n squared. Number of Statements Executed, n equals 10: 100. Number of Statements Executed, n equals 20: 400. Number of Statements Executed, n equals 1000: 10 to the power of 6. Number of Statements Executed, n equals 1 million: 10 to the power of 12. Row 5. Order of Magnitude: n cubed. Number of Statements Executed, n equals 10: 1000. Number of Statements Executed, n equals 20: 8000. Number of Statements Executed, n equals 1000: 10 to the power of 9. Number of Statements Executed, n equals 1 million: 10 to the power of 18. Row 6. Order of Magnitude: 2 to the power of n. Number of Statements Executed, n equals 10: 1024. Number of Statements Executed, n equals 20: Approximately 10 to the power of 6. Number of

Statements Executed, n equals 1000: Approximately 10 to the power of 300. Number of Statements Executed, n equals 1 million: Approximately 10 to the power of 3000000.

Back to Table

The table has 3 columns labelled f, left parentheses, n, right parentheses, Dominant Term, and Big Oh. The row entries are as follows. Row 1. f, left parentheses, n, right parentheses: 2 asterisk n plus 19. Dominant Term: 2 asterisk n. Big Oh: O, left parentheses, n, right parentheses. Row 2. f, left parentheses, n, right parentheses: 3 asterisk n squared plus 6 asterisk n plus 12. Dominant Term: 3 asterisk n squared. Big Oh: O, left parentheses, n squared, right parentheses. Row 3. f, left parentheses, n, right parentheses: n cubed plus 9 asterisk n squared plus 5 asterisk n plus 2. Dominant Term: n cubed. Big Oh: O, left parentheses, n cubed, right parentheses. Row 4. f, left parentheses, n, right parentheses: 3 asterisk 2 to the power of n plus 5 asterisk n cubed plus 3 asterisk n plus 7. Dominant Term: 3 asterisk 2 to the power of n. Big Oh: O, left parentheses, 2 to the power of n, right parentheses. Row 5. f, left parentheses, n, right parentheses: n plus 7 asterisk log n. Dominant Term: n. Big Oh: O, left parentheses, n, right parentheses. Row 6. f, left parentheses, n, right parentheses: 2 asterisk n asterisk log n plus 8 asterisk n plus log n plus 8. Dominant Term: 2 asterisk n asterisk log n. Big Oh: O, left parentheses, n asterisk log n, right parentheses. Row 7. f, left parentheses, n, right parentheses: 3 asterisk log n plus 35. Dominant Term: 3 asterisk log n. Big Oh: O, left parentheses, log n, right parentheses.

Back to Table

Program code. The code contains the following keywords: public, static, int, while, return. Line 1: public static int add Elements, left parentheses, int, left square bracket, right square bracket, a r r, right parentheses. Line 2: left brace. Line 3, indented once: int sum, equals, 0, semicolon. 2 forward slashes, left parentheses, 1, right parentheses. Line 4, indented once: int i equals 0, semicolon. 2 forward slashes, left parentheses, 2, right parentheses. Line 5, indented twice: while, left parentheses, i less than a r r dot length, right parentheses. 2 forward slashes, left parentheses, 3, right parentheses. Line 6, indented twice: left brace. Line 7, indented thrice: sum, plus, equals, a r r, left parentheses, i, right parentheses, semicolon. 2 forward slashes, left parentheses, 4, right parentheses. Line 8, indented thrice: i plus, plus, semicolon. 2 forward slashes, left parentheses, 5, right parentheses. Line 9, indented twice: right brace. Line 10, indented once: return sum, semicolon. 2 forward slashes, left parentheses, 6, right parentheses. Line 11: right brace.

Back to Example

Program code. The code contains the following keywords:

public, static, int, for, if, return. Line 1: public static int calculate Maximum, left parentheses, int, left square bracket, right square bracket, left square bracket, right square bracket, a r r, right parentheses. Line 2: left brace. Line 3, indented once: int maximum, equals, a r r, left square bracket, 0, right square bracket, left square bracket, 0, right square bracket, semicolon. 2 forward slashes, left parentheses, 1, right parentheses. Line 4, indented once: for, left parentheses, int i equals 0, semicolon, i less than a r r dot length, semicolon, i plus, plus, right parentheses. 2 forward slashes, left parentheses, 2, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: for, left parentheses, int j equals 0, semicolon, j less than a r r, left square bracket, i, right square bracket, dot, length, semicolon, j plus, plus, right parentheses. 2 forward slashes, left parentheses, 3, right parentheses. Line 7, indented twice: left brace. Line 8, indented thrice: if, left parentheses, maximum, less than, a r r, left square bracket, i, right square bracket, left square bracket, j, right square bracket, right parentheses. 2 forward slashes, left parentheses, 4, right parentheses. Line 9, indented 4 times: maximum, equals, a r r, left square bracket, i, right square bracket, left square bracket, j, right square bracket, semicolon. 2 forward slashes, left parentheses, 5, right parentheses. Line 10, indented twice: right brace. Line 11, indented once: right brace. Line 12, indented once: return maximum, semicolon. 2 forward slashes, left parentheses, 6, right parentheses. Line 13: right brace.

Back to Example

Program code. The code contains the following keywords: public, static, int, for, if, return. Line 1: public static int sequential Search, left parentheses, int, left square bracket, right square bracket, array, comma, int key, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int i equals 0, semicolon, i less than array dot length, semicolon, i plus, plus, right parenthess. 2 forward slashes, left parentheses, 1, right parentheses. Line 4, indented twice: if, left parentheses, array, left square bracket, i, right square bracket, 2 equals sign, key, right parentheses. 2 forward slashes, left parentheses, 2, right parentheses. Line 5, indented thrice: return i, semicolon. 2 forward slashes, left parentheses, 3, right parentheses. Line 6, indented once: return negative 1, semicolon. 2 forward slashes, left parentheses, 4, right parentheses. Line 7: right brace.

Back to Example

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: public static int power Of 2 A, left parentheses, int n, right parentheses. 2 forward slashes, n greater than or equal to 0. Line 2: left brace. Line 3, indented once: if, left parentheses, n, 2 equals sign, 0, right parentheses. Line 4, indented twice: return 1, semicolon. Line 5, indented once: else. Line 6, indented twice: return 2 asterisk power Of 2 A, left parentheses, n minus 1, right parentheses, semicolon.

Line 7: right brace.

Back to Example

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: public static int power Of 2 B, left parentheses, int n, right parentheses. 2 forward slashes, n greater than or equal to 0. Line 2: left brace. Line 3, indented once: if, left parentheses, n, 2 equals sign, 0, right parentheses. Line 4, indented twice: return 1, semicolon. Line 5, indented once: else. Line 6, indented twice: return power Of 2 B, left parentheses, n minus 1, right parentheses, plus, power Of 2 B, left parentheses, n minus 1, right parentheses, semicolon. Line 7: right.

Back to Example

Line 1: 2 forward slashes. 1. 5 asterisks. student writes this method. Line 2: Forward slash comma 2 asterisks. Sorts a r r in ascending order using the selection sort algorithm. Line 3: Asterisk. Adds a counter to count the number of statement executions. Line 4: Asterisk. Forward slash. Line 5: public int selection Sort comma left parentheses comma right parentheses. Line 6: Left brace. Line 7: 2 forward slashes. Note, colon, To count the number of statement executions comma use a counter. Line 8: 2 forward slashes. The variable counter has been declared and initialized for you. Line 9: 2 forward slashes. at the beginning of this method. Line 10: 2 forward slashes. Inside the body of the inner loop comma increment the counter. Line 11: 2 forward slashes. Replace the return statement so that this method returns the value of. Line 12: 2 forward slashes. the counter. To slow down or accelerate the animation comma modify the. Line 13: 2 forward slashes. argument of Pause dot wait in the handle method of the Array Animation Timer inner class. Line 14: int counter equals 0, semicolon. Line 15: int temp comma index Of Max, semicolon. Line 16: for left parentheses int i equals 0, semicolon. i less than a r r dot length minus 1, semicolon. i plus, plus right parentheses. Line 17: Left brace. Line 18: 2 forward slashes. find index of largest value in the subarray. Line 19: index Of Max equals 0, semicolon. Line 20: animate left parentheses i comma 0 comma 0 comma counter right parentheses, semicolon. Line 21: for left parentheses int j equals 1, semicolon. j less than a r r dot length minus i, semicolon. j plus, plus right parentheses. Line 22: Left brace. Line 23: if left parentheses a r r left square bracket j right square bracket greater than a r r left square bracket index Of Max right square bracket right parentheses. Line 24: index Of Max equals j, semicolon. Line 25: animate left parentheses i comma j comma index Of Max comma counter right parentheses, semicolon. Line 26: Right brace. Line 27: 2 forward slashes. swap a r r left square bracket index Of Max right square bracket and a r r left square bracket a r r dot length minus i minus 1 right square bracket. Line 28: temp equals a r r left square bracket index Of Max right square bracket , semicolon.

Line 29: a r r left square bracket index Of Max right square bracket equals a r r left square bracket a r r dot length minus i minus 1 right square bracket, semicolon. Line 30: a r r left square bracket a r r dot length minus i minus 1 right square bracket equals temp, semicolon. Line 31: Right brace. Line 32: animate left parentheses a r r dot length minus 1 comma 0 comma 0 comma counter right parentheses, semicolon. Line 33: return 0, semicolon. Line 34: Right brace. 2 forward slashes. end of selection Sort.

Back to Example

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: public static int recursive Binary Search. Line 2, indented multiple times: left parentheses, int, left square bracket, right square bracket, a r r, comma, int key, comma, int start, comma, int end, right parentheses. Line 3: left brace. Line 4, indented once: if, left parentheses, start, less than or equal to, end, right parentheses. Line 5, indented once: left brace. Line 6, indented once: 2 forward slashes, look at the middle element of the subarray. Line 7, indented once: int middle, equals, left parentheses, start plus end, right parentheses, forward slash, 2, semicolon. Line 8, indented once: if, left parentheses, a r r, left square bracket, middle, right square bracket, equals, equals, key, right parentheses. 2 forward slashes, found key, comma, base case. Line 9, indented twice: return middle, semicolon. Line 10, indented once: else if, left parentheses, a r r, left square bracket, middle, right square bracket, greater than, key, right parentheses. 2 forward slashes, look lower. Line 11, indented twice: return recursive Binary Search, left parentheses, a r r, comma, key, comma, start, comma, middle minus 1, right parentheses, semicolon. Line 12, indented once: else. 2 forward slashes, look higher. Line 13, indented twice: return recursive Binary Search, left parentheses, a r r, comma, key, comma, middle plus 1, comma, end, right parentheses, semicolon. Line 14, indented once: right brace. Line 15, indented once: else. 2 forward slashes, key not found, comma, base case. Line 16, indented twice: return minus 1, semicolon. Line 17: right brace.

Back to Example

The table has 2 columns labeled Number of Elements in Subarray As We Are Searching and Comparison Count. The row entries are as follows. Row 1. Number of Elements in Subarray As We Are Searching: 1024. Comparison Count: 1. Row 2. Number of Elements in Subarray As We Are Searching: 512. Comparison Count: 2. Row 3. Number of Elements in Subarray As We Are Searching: 256. Comparison Count: 3. Row 4. Number of Elements in Subarray As We Are Searching: 128. Comparison Count: 4. Row 5. Number of Elements in Subarray As We Are Searching: 64. Comparison Count: 5. Row 6. Number of Elements in Subarray As We Are Searching: 32. Comparison Count: 6. Row 7. Number of Elements in Subarray As We Are Searching: 16.

Comparison Count: 7. Row 8. Number of Elements in Subarray As We Are Searching: 8. Comparison Count: 8. Row 9. Number of Elements in Subarray As We Are Searching: 4. Comparison Count: 9. Row 10. Number of Elements in Subarray As We Are Searching: 2. Comparison Count: 10. Row 11. Number of Elements in Subarray As We Are Searching: 1. Comparison Count: 11.

Back to Table

Program code. The code contains the following keywords: public, static, void, int, for, while. Line 1: Forward slash. 2 asterisks. Performs an Insertion Sort on an integer array. Line 2: Asterisk. At symbol, param array array to sort. Line 3: Asterisk. Forward slash. Line 4: public static void insertion Sort, left parentheses, int, left square bracket, right square bracket, array, right parentheses. Line 5: left brace. Line 6, indented once: int j comma temp, semicolon. Line 7, indented once: for, left parentheses, int i equals 1, semicolon, i less than array dot length, semicolon, i plus, plus, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: j equals i, semicolon. Line 10, indented twice: temp equals array, left square bracket, i, right square bracket, semicolon. Line 11, indented twice: temp equals array, left square bracket, i, right square bracket, semicolon. Line 12, indented twice: while, left parentheses, j exclamation mark, equals, 0, 2 ampersands, array, left square bracket, j minus 1, right square bracket, greater than, temp, right parentheses.

Line 13, indented twice: left brace. Line 14, indented thrice: array, left square bracket, j, right square bracket, equals, array, left square bracket, j minus 1, right square bracket, semicolon. Line 15, indented thrice: j minus, minus, semicolon. Line 16, indented twice: right brace. Line 17, indented twice: array, left square bracket, j, right square bracket, equals, temp, semicolon. Line 18, indented once: right brace. Line 19: right brace.

Back to Example

Line 1: Forward slash. Asterisk. Simulation of various sorting algorithms. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot asterisk, semicolon. Line 6: blank. Line 7: public class Simulation. Line 8: Left brace. Line 9: public static int SIZE equals 1024, semicolon. Line 10: public static int count equals 0, semicolon. Line 11: blank. Line 12: public static void main, left parentheses, String left square bracket, right square bracket a r g s, right parentheses. Line 13: Left brace. Line 14: int left square bracket, right square bracket grades equals generate Array, left parentheses, SIZE, right parentheses, semicolon. Line 15: blank. Line 16: 2 forward slashes. sort unsorted array using insertion sort. Line 17: insertion Sort, left parentheses, grades, right parentheses, semicolon. Line 18: output, left parentheses, grades, right parentheses, semicolon. Line 19: System dot out dot print l n, left parentheses, double quotes Unsorted array, colon, double quotes, plus. Line 20: double quotes Insertion sort count equals double quotes plus count, right parentheses, semicolon. Line 21: blank. Line 22: 2 forward slashes. sort sorted array using insertion sort. Line 23: count equals 0, semicolon. Line 24: insertion Sort, left parentheses, grades , right parentheses , semicolon. Line 25: output, left parentheses, grades , right parentheses, semicolon. Line 26: System dot out dot print l n, left parentheses, double quotes Sorted array, colon, double quotes, plus. Line 27: double quotes Insertion sort count equals double quotes plus count, right parentheses, semicolon. Line 28: blank. Line 29: grades equals generate Array, left parentheses, SIZE, right parentheses, semicolon. Line 30: 2 forward slashes. sort unsorted array using merge sort. Line 31: count equals 0, semicolon. Line 32: merge Sort, left parentheses, grades, 0, grades dot length minus 1, right parentheses, semicolon. Line 33: output, left parentheses, grades, right parentheses, semicolon. Line 34: System dot out dot print l n, left parentheses, double quotes Unsorted array, colon, double quotes, plus. Line 35: double quotes Merge sort count equals double quotes plus count, right parentheses, semicolon. Line 36: blank. Line 37: 2 forward slashes. sort sorted array using merge sort. Line 38: count equals 0, semicolon. Line 39: merge Sort, left parentheses, grades, 0, grades dot length minus 1, right parentheses, semicolon. Line 40: output, left parentheses, grades, right parentheses, semicolon. Line 41: System dot out dot print l n, left parentheses, double quotes Sorted array, colon, double quotes plus. Line 42: double quotes Merge sort count equals double quotes plus count, right parentheses, semicolon. Line 43: blank. Line 44: grades equals generate Array, left parentheses, SIZE, right parentheses, semicolon. Line 45: 2 forward slashes, sort unsorted array using quick sort. Line 46: count equals 0, semicolon. Line 47: quick Sort, left parentheses, grades, 0, grades dot length minus 1, right parentheses, semicolon. Line 48: output, left parentheses, grades, right parentheses, semicolon. Line 49: System dot out dot print l n, left parentheses, double quotes Unsorted array, colon, double quotes, plus. Line 50: double quotes Quick sort count equals double quotes plus count, right parentheses, semicolon. Line 51: blank. Line 52: 2 forward slashes. sort sorted array using quick sort. Line 53: count equals 0, semicolon. Line 54: quick Sort, left parentheses, grades, 0, grades dot length minus 1, right parentheses, semicolon. Line 55: output, left parentheses, grades, right parentheses, semicolon. Line 56: System dot out dot print l n, left parentheses, double quotes Sorted array, colon, double quotes plus. Line 57: double quotes Quick sort count equals double quotes plus count, right parentheses, semicolon. Line 58: Right brace. Line 59: blank. Line 60: Forward slash. Asterisk. Asterisk. Line 61: asterisk output method, colon, outputs a r r. Line 62: asterisk at symbol param a r r, an array of ints. Line 63: Asterisk. Forward slash. Line 64: public static void output, left parentheses, int left square bracket, right square bracket a r r, right parentheses. Line 65: Left brace. Line 66: for, left parentheses, int i equals 0, semicolon. i less than a r r dot length minus 1, semicolon.

i, plus, plus, right parentheses. Line 67: System dot out dot print, left parentheses, a r r left square bracket i right square bracket plus double quotes double quotes, right parentheses, semicolon. Line 68: System dot out dot print l n, left parentheses, right parentheses, semicolon. Line 69: Right brace. Line 70: blank. Line 71: Forward slash. Asterisk. Asterisk. Line 72: asterisk generate Array method, colon, sorts generates an array of random ints. Line 73: asterisk at symbol param n, an int greater than 0. Line 74: asterisk at symbol return an array of n random integers. Line 75: Asterisk. Forward slash. Line 76: public static int left square bracket, right square bracket generate Array, left parentheses, int n, right parentheses. Line 77: Left brace. Line 78: int left square bracket, right square bracket a r r equals new int left square bracket n right square bracket, semicolon. Line 79: Random rand equals new Random, left parentheses, right parentheses, semicolon. Line 80: int number equals 0, semicolon. Line 81: for, left parentheses, int i equals 0, semicolon. i less than n, semicolon. i, plus, plus, right parentheses. Line 82: Left brace. Line 83: number equals rand dot next Int, left parentheses, 100000, right parentheses, semicolon. Line 84: a r r left square bracket i right square bracket equals number, semicolon. Line 85: Right brace. Line 86: return a r r, semicolon. Line 87: Right brace. Line 88: blank. Line 89: Forward slash. Asterisk. Asterisk. Line 90: asterisk insertion Sort method, colon, sorts a r r in ascending order. Line 91: asterisk at symbol param a r r, an array of ints. Line 92: Asterisk. Forward slash. Line 93:

public static void insertion Sort, left parentheses, int left square bracket, right square bracket a r r, right parentheses. Line 94: Left brace. Line 95: int j, temp, semicolon. Line 96: for, left parentheses, int i equals 1, semicolon, i less than a r r dot length, semicolon, i, plus, plus, right parentheses. Line 97: Left brace. Line 98: j equals i, semicolon. Line 99: temp equals a r r left square bracket i right square bracket, semicolon. Line 100: count, plus, plus, semicolon. Line 101: while, left parentheses, j, exclamation mark, equals, 0, 2 ampersands a r r left square bracket, j minus 1, right square bracket, greater than, temp, right parentheses. Line 102: Left brace. Line 103: count, plus, plus. , semicolon. Line 104: a r r left square bracket j right square bracket equals a r r left square bracket j minus 1 right square bracket, semicolon. Line 105: j minus minus, semicolon. Line 106: Right brace. Line 107: a r r left square bracket j right square bracket equals temp, semicolon. Line 108: Right brace. Line 109: Right brace. Line 110: blank. Line 111: Forward slash. Asterisk. Asterisk. Line 112: asterisk merge Sort method, colon, sorts the subarray of a r r. Line 113: asterisk, left parentheses, from index start to index end, both included, right parentheses. Line 114: asterisk in ascending order. Line 115: asterisk at symbol param a r r, an array of ints. Line 116: asterisk at symbol param start an int, the starting index of the subarray. Line 117: asterisk at symbol param end an int, the ending index of the subarray. Line 118: Asterisk. Forward slash. Line 119: public static void merge Sort, left parentheses, int left square bracket, right square bracket a r r, int start, int end, right parentheses. Line 120: Left brace. Line 121: if, left parentheses, start less than end, right parentheses, 2 forward slashes, general case. Line 122: Left brace. Line 123: int middle equals, left parentheses, start plus end, right parentheses, forward slash, 2, semicolon. Line 124: 2 forward slashes, sort the left part of the array. Line 125: merge Sort, left parentheses, a r r, start, middle, right parentheses, semicolon. Line 126: 2 forward slashes, sort the right part of the array. Line 127: merge Sort, left parentheses, a r r, middle plus 1, end, right parentheses, semicolon. Line 128: 2 forward slashes, merge the 2 parts. Line 129: merge, left parentheses, a r r, start, middle, middle plus 1, end, right parentheses, semicolon. Line 130: Right brace. Line 131: 2 forward slashes, else, base case, empty or 1 element array, already sorted. Line 132: Right brace. Line 133: blank. Line 134: Forward slash. Asterisk. Asterisk. Line 135: asterisk merge method, colon, merges, in ascending order. Line 136: asterisk the two subarrays of a r r defined as follows, colon. Line 137: asterisk left subarray, colon, between indexes start 1 and end 1, both included. Line 138: asterisk right subarray, colon, between indexes start 2 and end 2, both included. Line 139: asterisk at symbol param a r r, an array of ints. Line 140: asterisk at symbol param start 1 an int, the starting index of the left subarray. Line 141: asterisk, at symbol, param end 1 an int, the ending index of the left subarray. Line 142: asterisk at symbol param start 2 an int, the starting index of the right subarray. Line 143: asterisk at symbol param end 2 an int, the ending index of the right subarray. Line 144: Asterisk. Forward slash. Line 145: public static void merge, left parentheses, int left square bracket, right square bracket a r r, int start 1, int end 1. Line 146: int start 2, int end 2, right parentheses. Line 147: Left brace. Line 148: 2 forward slashes, merge the 2 sides into a temp array. Line 149: int left square bracket, right square bracket, temp equals new int left square bracket, a r r dot length right square bracket, semicolon, 2 forward slashes, temporary array. Line 150: int i equals start 1, semicolon. Line 151: int j equals start 2, semicolon. Line 152: int k equals start 1, semicolon. Line 153: blank. Line 154: 2 forward slashes, generate merged subarray until one side is done. Line 155: while, left parentheses, i less than or equal to end 1, 2 ampersands, j less than or equal to end 2, right parentheses. Line 156: Left brace. Line 157: count, plus, plus, semicolon. Line 158: if, left parentheses, a r r, left square bracket, i, right square bracket, less than a r r left square bracket, j right square bracket, right parentheses. Line 159: Left brace. Line 160: temp left square bracket k right square bracket equals a r r left square bracket, i right square bracket, semicolon. Line 161: i, plus, plus, semicolon. Line 162: Right brace. Line 163: else. Line 164: Left brace. Line 165: temp left square bracket k right square bracket equals a r r left square bracket j right square bracket, semicolon. Line 166: j, plus, plus, semicolon. Line 167: Right brace. Line 168: k, plus, plus, semicolon. Line 169: Right brace. Line 170: blank. Line

171: 2 forward slashes, flush left subarray if necessary. Line 172: while, left parentheses, i less than or equal to end 1, right parentheses. Line 173: Left brace. Line 174: temp left square bracket k right square bracket equals a r r left square bracket i right square bracket, semicolon. Line 175: i, plus, plus, semicolon. Line 176: k, plus, plus, semicolon. Line 177: Right brace. Line 178: blank. Line 179: 2 forward slashes, flush right subarray if necessary. Line 180: while, left parentheses, j less than or equal to end 2, right parentheses. Line 181: Left brace. Line 182: temp left square bracket k right square bracket equals a r r left square bracket j right square bracket, semicolon. Line 183: j, plus, plus, semicolon. Line 184: k, plus, plus, semicolon. Line 185: Right brace. Line 186: blank. Line 187: 2 forward slashes, write back temp into a r r. Line 188: for, left parentheses, i equals start 1, semicolon. i less than or equal to end 2, semicolon. i, plus, plus, right parentheses. Line 189: Left brace. Line 190: a r r left square bracket, i right square bracket, equals temp left square bracket i right square bracket, semicolon. Line 191: Right brace. Line 192: Right brace. Line 193: blank. Line 194: Forward slash. Asterisk. Asterisk. Line 195: asterisk quick Sort method, colon, sorts the subarray of a r r. Line 196: asterisk, left parentheses, from index start to index end, both included, right parentheses. Line 197: asterisk, in ascending order. Line 198: asterisk at symbol, param a r r, an array of ints. Line 199: asterisk at symbol, param start an int, the starting index of the subarray. Line 200: asterisk at symbol, param end an int, the ending index of the subarray. Line 201: Asterisk. Forward slash. Line 202: public static void quick Sort, left parentheses, int left square bracket, right square bracket a r r, int start, int end, right parentheses. Line 203: Left brace. Line 204: if, left parentheses, end greater than start, right parentheses, 2 forward slashes, general case greater than or equal to 2 elements. Line 205: Left brace. Line 206: int pivot Index equals partition, left parentheses, a r r, start, end, right parentheses, semicolon. Line 207: 2 forward slashes. quick sort left subarray. Line 208: quick Sort, left parentheses, a r r, start, pivot Index minus 1, right parentheses, semicolon. Line 209: 2 forward slashes, quick sort right subarray. Line 210: quick Sort, left parentheses, a r r, pivot Index plus 1, end, right parentheses, semicolon. Line 211: Right brace. Line 212: 2 forward slashes, else, 0 or 1 element, equals, equals, right angled bracket, subarray is sorted, do nothing. Line 213: Right brace. Line 214: blank. Line 215: Forward slash. Asterisk. Asterisk. Line 216: asterisk partition method, colon, partitions the subarray of a r r. Line 217: asterisk, left parentheses, from index start to index end, both included, right parentheses. Line 218: asterisk around the pivot, a r r left square bracket start right square bracket. Line 219: asterisk at symbol, param a r r, an array of ints. Line 220: asterisk at symbol, param start an int, the starting index of the subarray. Line 221: asterisk at symbol, param end an int, the ending index of the subarray. Line 222: Asterisk. Forward slash. Line 223: public static int partition, left parentheses, int left square bracket, right square bracket a r r, int start, int end, right parentheses. Line 224: Left brace. Line 225: int j equals start, semicolon. Line 226: int pivot equals a r r left square bracket start right square bracket, semicolon. Line 227: for, left parentheses, int i equals start plus 1, semicolon. i less than or equal to end, semicolon. i, plus, plus, right parentheses. Line 228: Left brace. Line 229: count, plus, plus, semicolon. Line 230: if, left parentheses, a r r left square bracket i right square bracket less than pivot, right parentheses. Line 231: Left brace. Line 232: j, plus, plus, semicolon. Line 233: swap, left parentheses, a r r, i, j, right parentheses, semicolon. Line 234: Right brace. Line 235: Right brace. Line 236: swap, left parentheses, a r r, start, j, right parentheses, semicolon. Line 237: return j, semicolon. Line 238: Right brace. Line 239: blank. Line 240: Forward slash. Asterisk. Asterisk. Line 241: asterisk swap method, colon, swaps the elements of a r r. Line 242: asterisk at index i and index j. Line 243: asterisk at symbol, param a r r, an array of ints. Line 244: asterisk at symbol, param i an int, a valid index of a r r. Line 245: asterisk at symbol, param j an int, a valid index of a r r. Line 246: Asterisk. Forward slash. Line 247: public static void swap, left parentheses, int left square bracket, right square bracket a r r, int i, int j , right parentheses. Line 248: Left brace. Line 249: int temp equals a r r left square bracket i right square bracket, semicolon. Line 250: a r r left square bracket i right square bracket equals a r r left square bracket j right square bracket , semicolon. Line 251: a r r left square bracket j right square bracket equals temp, semicolon. Line 252: Right brace. Line 253: Right brace.

Back to Example

The table has 4 columns labeled Algorithm, Starting Array, Comparisons, left parentheses, simulation, right parentheses, and For n equals 1024. The row entries are as follows. Row 1. Algorithm: Insertion sort. Starting Array: Unsorted. Comparisons, left parentheses, simulation, right parentheses: 259873. For n equals 1024: About n asterisk, left parentheses, n minus 1, right parentheses, forward slash, 4. Row 2. Algorithm: Insertion sort. Starting Array: Sorted. Comparisons, left parentheses, simulation, right parentheses: 1023. For n equals 1024: n minus 1. Row 3. Algorithm: Merge sort. Starting Array: Unsorted. Comparisons, left parentheses, simulation, right parentheses: 8917. For n equals 1024: About n asterisk, log n. Row 4. Algorithm: Merge sort. Starting Array: Sorted. Comparisons, left parentheses, simulation, right parentheses: 5120. For n equals 1024: About n asterisk log n forward slash 2. Row 5. Algorithm: Quick sort. Starting Array: Unsorted. Comparisons, left parentheses, simulation, right parentheses: 10332. For n equals 1024: About n asterisk log n. Row 6. Algorithm: Quick sort. Starting Array: Sorted. Comparisons, left parentheses, simulation, right parentheses: 523776. For n equals 1024: n asterisk, left parentheses, n minus 1, right parentheses, forward slash, 2.

Back to Table

H T M L code for a simple web page. Program code. In the code, the words in the variable names are merged. Line 1: Left angle bracket, h t m l, right angle bracket. Line 2, indented once: left angle bracket, head, right angle bracket. Line 3, indented twice: Left angle bracket, title, right angle bracket, My web page, left angle bracket, forward slash, title, right angle bracket. Line 4, indented once: Left angle bracket, forward slash, head, right angle bracket. Line 5, indented once: Left angle bracket, body, right angle bracket. Line 6, indented twice: Left angle bracket, p, right angle bracket, Web tools, left angle bracket forward slash, p, right angle bracket. Line 7, indented twice: Left angle bracket, o l, right angle bracket. Line 8, indented 3 times: Left angle bracket, l i, right angle bracket, H t m l, left angle bracket, forward slash, l i, right angle bracket. Line 9, indented 3 times: Left angle bracket, l i, right angle bracket, C s s, left angle bracket, forward slash, l i, right angle bracket. Line 10, indented 3 times: Left angle bracket, l i, right angle bracket, JavaScript, left angle bracket, forward slash, l i, right angle bracket. Line 11, indented twice: Left angle bracket, forward slash, o l, right angle bracket. Line 12, indented once: Left angle bracket, forward slash, body, right angle bracket. Line 13: Left angle bracket, forward slash, h t m l, right angle bracket.

Back to Example

A table detailing tree terminology. The table has 8 rows and 2 columns. It has the following column headings. Term. Description. The row details in the table are the following. Row 1: Term, Leaf; Description, A node with no child. Row 2: Term, Interior node; Description, A node with at least one child. Row 3: Term, Siblings; Description, Two (or more) nodes with the same parent. Row 4: Term, Degree of a node; Description, The number of children of a node. Row 5: Term, Depth of a node; Description, The number of edges on the path from the root to the node. Row 6: Term, Level of a node; Description, The number of edges on the path from the root to the node. Row 7: Term, Height of a node; Description, The largest number of edges on a path from a node to a leaf. Row 8: Term, Height of the tree; Description, The height of the root.

Back to Table

A table detailing number of nodes per level in a full binary tree. The table has 8 rows and 2 columns. It has the following column headings. Level. Number of Nodes. The row details in the table are the following. Row 1: Level, 0; Number of Nodes, 1 equals 2 to the power of 0. Row 2: Level, 1; Number of Nodes, 2 equals 2 to the power of 1. Row 3: Level, 2; Number of Nodes, 4 equals 2 to the power of 2. Row 4: Level, 3; Number of Nodes, 8 equals 2 to the power of 2. Row 5: Level, 4; Number of Nodes, 16 equals 2 to the power of 2. Row 6: Level, 5; Number of Nodes, 32 equals 2 to the power of 5. Row 7: Level, 6; Number of Nodes, 64 equals 2 to the power of 6. Row 8:

Level, L; Number of Nodes, 2 to the power of L.

Back to Table

Code for the Integer Node class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, The Integer Node class. Line 2: Anderson, comma, Franceschi. Line 3: Forward slash, asterisk. Line 4: Blank. Line 5: public, class, Integer Node. Line 6: Left brace. Line 7, indented once: private, i n t, data, semicolon. Line 8, indented once: private, Integer Node, left, semicolon. Line 9, indented once: private, Integer Node, right, semicolon. Line 10: Blank. Line 11, indented once: Forward slash, asterisk, asterisk, Constructor. Line 12, indented once: asterisk, at, param, data. Line 13, indented once: Asterisk, initializes data instance variable with data. Line 14, indented once: Asterisk, forward slash. Line 15, indented once: public, Integer Node, left parenthesis, i n t, data, right parenthesis. Line 16, indented once: Left brace. Line 17, indented twice: set Data, left parenthesis, data, right parenthesis, semicolon. Line 18, indented once: right brace. Line 19: Blank. Line 20, indented once: Forward slash, asterisk, asterisk, Accessor for data. Line 21, indented once: asterisk, at return data. Line 22, indented once: asterisk, forward slash. Line 23, indented once: public, i n t, get Data, left parenthesis, right parenthesis. Line 24, indented once: Left brace. Line 25, indented twice: return, data, semicolon. Line 26, indented once: right brace. Line 27: Blank. Line 28, indented once:

Forward slash, asterisk, asterisk, Accessor for left. Line 29, indented once: asterisk, at, return, left. Line 30, indented once: asterisk, forward slash. Line 31, indented once: public, Integer Node, get Left, left parenthesis, right parenthesis. Line 32, indented once: Left brace. Line 33, indented twice: return, left, semicolon. Line 34, indented once: Right brace. Line 35: Blank. Line 36, indented once: Forward slash, asterisk, asterisk, Accessor for right. Line 37, indented once: Asterisk, at, return, right. Line 38, indented once: Asterisk, forward slash. Line 39, indented once: public, Integer Node, get Right, left parenthesis, right parenthesis. Line 40, indented once: Left brace. Line 41, indented twice: return, right, semicolon. Line 42, indented once: Right brace. Line 43: Blank. Line 44, indented once: Forward slash, asterisk, asterisk, Mutator for data. Line 45, indented once. Asterisk, at, param, data, tab space, an i n t, comma, the new value for data. Line 46, indented once: Asterisk, at, return, tab space, a reference to this object. Line 47, indented once: asterisk, forward slash. Line 48, indented once: public, Integer Node, set Data, left parenthesis, i n t, data, right parenthesis. Line 49, indented once: Left brace. Line 50, indented twice: this, dot, data, equals, data, semicolon. Line 51, indented twice: return, this, semicolon. Line 52, indented once: right brace. Line 53: Blank. Line 54, indented once: Forward slash, asterisk, asterisk, Mutator for left. Line 55, indented once: Asterisk, @ param, left a Node, comma, the new Node reference for left. Line 56, indented once: *, @, return, tab space, a reference to this object. Line 57, indented once: asterisk, forward slash. Line 58, indented once: public, Integer Node, set Left, left parenthesis, Integer Node, left, right parenthesis. Line 59, indented once: Left brace. Line 60, indented twice: this, dot, left, equals, left, semicolon. Line 61, indented twice: return, this, semicolon. Line 62, indented once: Right brace. Line 63: Blank. Line 64, indented once: Forward slash, asterisk, asterisk, Mutator for right. Line 65, indented once: asterisk, @ param, right a Node, comma, the new Node reference for right. Line 66, indented once: asterisk, @ return, tab space, a reference to this object. Line 67, indented once: asterisk, forward slash. Line 68, indented once: public, Integer Node, set Right, left parenthesis, Integer Node, right, right parenthesis. Line 69, indented once: Left brace. Line 70, indented twice: this, dot, right, equals, right, semicolon. Line 71, indented twice: return, this, semicolon. Line 72, indented once: Right brace. Line 73: Right brace.

Back to Example

Code for the Binary Tree class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, The Binary Tree class. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: public, class, Binary Tree. Line 6: Left brace. Line 7, indented once: private, Integer Node, root, semicolon. Line 8: Blank. Line 9, indented once: Forward slash, asterisk, asterisk, Constructor. Line 10, indented once: asterisk, at, param, value an i n t. Line 11, indented once: asterisk, constructs a binary tree with one node storing value. Line 12, indented once: asterisk, forward slash. Line 13, indented once: public, Binary Tree, left parenthesis, i n t, value, right parenthesis. Line 14, indented once: Left brace. Line 15, indented twice: root, equals, new, Integer Node, left parenthesis, value, right parenthesis, semicolon. Line 16, indented once: right brace. Line 17: Blank. Line 18, indented once: Forward slash, asterisk, asterisk, attach Left method. Line 19, indented once: asterisk, at, param, b t, a Binary Tree reference. Line 20, indented once: asterisk, attaches b t as the left subtree of this Binary Tree. Line 21, indented once: Asterisk, forward slash. Line 22, indented once: public, void, attach Left, left parenthesis, Binary Tree, b t, right parenthesis. Line 23, indented once: Left brace. Line 24, indented twice: root, dot, set Left, left parenthesis, b t, dot, root, right parenthesis, semicolon. Line 25, indented once: right brace. Line 26: Blank. Line 27, indented once: Forward slash, asterisk, asterisk, attach Right method. Line 28, indented once: asterisk, at, param, b t a Binary Tree reference. Line 29, indented once: asterisk, attaches b t as the right subtree of this Binary Tree. Line 30, indented once: Asterisk, forward slash. Line 31, indented once: public, void, attach Right, left parenthesis, Binary Tree, b t, right parenthesis. Line 32, indented once: Left brace. Line 33, indented twice: root, dot, set Right, left parenthesis, b t, dot, root, right parenthesis, semicolon. Line 34, indented once: Right brace. Line 35: Blank. Line

36, indented once: Forward slash, asterisk, asterisk, pre Order method. Line 37, indented once: asterisk, @, return, a String representation of the pre order. Line 38, indented once: asterisk, traversal of this Binary Tree. Line 39, indented once: asterisk, forward slash. Line 40, indented once: public, String, pre Order, left parenthesis, right parenthesis. Line 41, indented once: Left brace. Line 42, indented twice: return, pre Order, left parenthesis, root, right parenthesis, semicolon. Line 43, indented once: right brace. Line 44: Blank. Line 45, indented once: Forward slash, asterisk, asterisk, pre Order recursive method. Line 46, indented once: asterisk, at, param, n d, an Integer Node reference. Line 47, indented once: asterisk, at return, a String representation of the pre order. Line 48, indented once: asterisk, tab space, traversal of the Binary Tree "rooted" at n d. Line 49, indented once: asterisk, forward slash. Line 50, indented once: private, String, pre Order, left parenthesis, Integer Node, n d, right parenthesis. Line 51, indented once: Left brace. Line 52, indented twice: if, left parenthesis, n d, not equals, null, right parenthesis. Line 53, indented twice: Left brace. Line 54, indented 3 times: forward slash, forward slash, visit n d. Line 55, indented 3 times: String, result, equals, n d, dot, get Data, left parenthesis, right parenthesis, plus, open quotes, space, close quotes, semicolon. Line 56, indented 3 times: Forward slash, forward slash, visit left subtree in pre order. Line 57, indented 3 times: result, plus, equals, pre Order, left parenthesis, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 58, indented 3 times: forward slash, forward slash, visit right subtree in pre order. Line 59, indented 3 times: result, plus, equals, pre Order, left parenthesis, n d, dot, get Right, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 60, indented 3 times: return, result, semicolon. Line 61, indented twice: Right brace. Line 62, indented twice: else. Line 63, indented once: return, open quotes, slow quotes, semicolon. Line 64, indented once: Right brace. Line 65: Blank. Line 66, indented once: Forward slash, asterisk, asterisk, in Order method. Line 67, indented once: asterisk, at, return a String representation of the in order. Line 68, indented once: asterisk, traversal of this Binary Tree. Line 69, indented once: asterisk, forward slash. Line 70, indented once: public, String, in Order, left parenthesis, right parenthesis. Line 71, indented once: Left brace. Line 72, indented twice: return, in Order, left parenthesis, root, right parenthesis, semicolon. Line 73, indented once: right brace. Line 74: Blank. Line 75, indented once: Forward slash, asterisk, asterisk, in Order recursive method. Line 76, indented once: asterisk, @, param, n d an Integer Node reference. Line 77, indented once: asterisk, at, return a String representation of the in order. Line 78, indented once: asterisk, traversal of the Binary Tree "rooted" at n d. Line 79, indented once: asterisk, forward slash. Line 80, indented once: private, String, in Order, left parenthesis, Integer Node, n d, right parenthesis. Line 81, indented once: Left brace. Line 82, indented twice: if, left parenthesis, n d, not equals, null, right parenthesis. Line 83, indented twice: Left brace. Line 84, indented 3 times: forward slash, forward slash, visit left subtree in order. Line 85, indented 3 times: String, result, equals, in Order, left parenthesis, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 86, indented 3 times: forward slash, forward slash, visit n d. Line 87, indented 3 times: result, plus, equals, n d, dot, get Data, left parenthesis, right parenthesis, plus, open quotes, space, close quotes, semicolon. Line 88, indented 3 times: forward slash, forward slash, visit right subtree in order. Line 89, indented 3 times: result, plus, equals in Order, left parenthesis, n d, dot, get Right, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 90, indented 3 times: return, result, semicolon. Line 91, indented twice: Right brace. Line 92, indented twice: else. Line 93, indented 3 times: return, open quotes, close quotes, semicolon. Line 94, indented once: Right brace. Line 95: Blank. Line 96, indented once: Forward slash, asterisk, asterisk, post Order method. Line 97, indented once: Asterisk, at, return, a String representation of the post order. Line 98, indented once: asterisk, traversal of this Binary Tree. Line 99, indented once: Asterisk, forward slash. Line 100, indented once: public, String, post Order, left parenthesis, right parenthesis. Line 101, indented once: Left brace. Line 102, indented twice: return post Order, left parenthesis, root, right parenthesis, semicolon. Line 103, indented once: right brace. Line 104: Blank. Line 105, indented once: Forward slash, asterisk, asterisk, post Order recursive method. Line 106, indented once: Asterisk, at, param, n d, an Integer Node reference. Line 107, indented once: Asterisk, at, return, a String representation of the post order. Line 108, indented once: asterisk, traversal of the Binary Tree "rooted" at n d. Line 109, indented once: Asterisk, forward slash. Line 110, indented once: private, String, post Order, left parenthesis, Integer Node, n d, right parenthesis. Line 111, indented once: Left brace. Line 112, indented twice: if, left parenthesis, n d, not equals, null, right parenthesis. Line 113, indented twice: Left brace. Line 114, indented 3 times: forward slash, forward slash, visit left subtree in order. Line 115, indented 3 times: String, result, equals, post Order, left parenthesis, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 116, indented 3 times: Forward slash, forward slash, visit right subtree in order. Line 117, indented 3 times: result, plus, equals, post Order, left parenthesis, n d, dot, get Right, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 118, indented 3 times: Forward slash, forward slash, visit n d. Line 119, indented 3 times: result, plus, equals, n d, dot, get Data, left parenthesis, right parenthesis, plus, open quotes, space, close quotes, semicolon. Line 120, indented 3 times: return, result, semicolon. Line 121, indented twice: Right brace. Line 122, indented once: else. Line 123, indented once: return, open quotes, close quotes, semicolon. Line 124, indented once: Right brace. Line 125: Right brace.

Back to Example

A table detailing tree traversal strategies and results for the binary tree in figure 15.10. The table has 4 rows and 3 columns. It has the following column headings. Traversal. Strategy. Result. The row details are the following. Row 1: Traversal, Level Order; Strategy, Visit one level at a time, left to right; Result, 33, 20, 21, 34, 87, 84, 17, 25, 44. Row 2: Traversal, Pre Order; Strategy, Visit node the first time we see it; Result, 33, 20, 34, 17, 87, 25, 44, 21, 84. Row 3: Traversal, In Order; Strategy, Visit node the second time we see it; Result, 34, 17, 20, 25, 87, 44, 33, 21, 84. Row 4: Traversal, Post Order; Strategy, Visit node the third or last time we see it; Result, 17, 34, 25, 44, 87, 20, 84, 21, 33.

Back to Table

Code for the Binary Tree Client class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, The Binary Tree Client class. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: public, class, Binary Tree Client. Line 5: Left brace. Line 6, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 7, indented once: Left brace. Line 8, indented twice: Binary Tree, b t 33, equals, new, Binary Tree, left parenthesis, 33, right parenthesis, semicolon. Line 9, indented twice: Binary Tree, b t 20, equals, new, Binary Tree, left parenthesis, 20, right parenthesis, semicolon. Line 10, indented twice: Binary Tree, b t 21, equals, new, Binary Tree, left parenthesis, 21, right parenthesis, semicolon. Line 11, indented twice: Binary Tree, b t 34, equals, new, Binary Tree, left parenthesis, 34, right parenthesis, semicolon. Line 12, indented twice: Binary Tree, b t 87, equals, new, Binary Tree, left parenthesis, 87, right parenthesis, semicolon. Line 13, indented twice: Binary Tree, b t 84, equals, new, Binary Tree, left parenthesis, 84, right parenthesis, semicolon. Line 14, indented twice: Binary Tree, b t 17, equals, new, Binary Tree, left parenthesis, 17, right parenthesis, semicolon. Line 15, indented twice: Binary Tree, b t 25, equals, new, Binary Tree, left parenthesis, 25, right parenthesis, semicolon. Line 16, indented twice: Binary Tree, b t 44, equals, new, Binary Tree, left parenthesis, 44, right parenthesis, semicolon. Line 17: Blank. Line 18, indented twice: b t 33, dot, attach Left, left parenthesis, b t 20, right parenthesis, semicolon. Line 19, indented twice: b t 33, dot, attach Right, left parenthesis, b t 21, right parenthesis, semicolon. Line 20: Blank. Line 21, indented twice: b t 20, dot, attach Left, left parenthesis, b t 34, right parenthesis, semicolon. Line 22, indented twice: b t 20, dot, attach Right, left parenthesis, b t 87, right parenthesis, semicolon. Line 23, indented twice: b t 21, dot, attach Right, left parenthesis, b t 84, right parenthesis, semicolon. Line 24: Blank. Line 25, indented twice: b t 34, dot, attach Right, left parenthesis, b t 17, right parenthesis, semicolon. Line 26, indented twice: b t

87, dot, attach Left, left parenthesis, b t 25, right parenthesis, semicolon. Line 27, indented twice: b t 87, dot, attach Right, left parenthesis, b t 44, right parenthesis, semicolon. Line 28: Blank. Line 29, indented twice: System, dot, out, dot, print I n, left parenthesis, open quotes, Pre order, colon, close quotes, plus, b t 33, dot, pre Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 30, indented twice: System, dot, out, dot, print I n, left parenthesis, open quotes, In order, colon, close quotes, plus, b t 33, dot, in Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 31, indented twice: System, dot, out, dot, print I n, left parenthesis, open quotes, Post order, colon, close quotes, plus, b t 33, dot, post Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 32, indented twice: Right brace. Line 33: Right brace.

Back to Example

Code for the Binary Tree class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, The Binary Search Tree class. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: public, class, Binary Search Tree. Line 6: Left brace. Line 7, indented once: private, Integer Node, root, semicolon. Line 8: Blank. Line 9, indented once: Forward slash, asterisk, asterisk, default constructor. Line 10, indented once: Asterisk, constructs an empty Binary Search Tree.

Line 11, indented once: asterisk, forward slash. Line 12, indented once: public, Binary Search Tree, left parenthesis, right parenthesis. Line 13, indented once: Left brace. Line 14, indented twice: root, equals, null, semicolon. Line 15, indented once: Right brace. Line 16: Blank. Line 17, indented once: Forward slash, asterisk, asterisk, insert method. Line 18, indented once: asterisk, at, param, value an i n t. Line 19, indented once: Asterisk, inserts value as a leaf in this B S T. Line 20, indented once: Asterisk, forward slash. Line 21, indented once: public, void, insert, left parenthesis, i n t, value, right parenthesis. Line 22, indented once: Left brace. Line 23, indented twice: if, left parenthesis, root, equals, equals, null, right parenthesis. Line 24, indented 3 times: root, equals, new, Integer Node, left parenthesis, value, right parenthesis, semicolon. Line 25, indented twice: else. Line 26, indented 3 times: insert, left parenthesis, value, comma, root, right parenthesis, semicolon. Line 27, indented once: Right brace. Line 28: Blank. Line 29, indented once: Forward slash, asterisk, asterisk, insert recursive method. Line 30, indented once: asterisk, at, param, value an i n t. Line 31, indented once: asterisk, at, param, n d an Integer Node reference. Line 32, indented once: asterisk, inserts value in the B S T "rooted" at n d. Line 33, indented once: Asterisk, forward slash. Line 34, indented once: private, void, insert, left parenthesis, i n t, value, comma, Integer Node, n d, right parenthesis. Line 35, indented once: Left brace. Line 36, indented twice: if, left parenthesis, value, less than, n d, dot, get Data, left

parenthesis, right parenthesis, right parenthesis, forward slash, forward slash, insert somewhere left. Line 37, indented twice: Left brace. Line 38, indented 3 times: if, left parenthesis, n d, dot, get Left, left parenthesis, right parenthesis, equals, equals, null, right parenthesis, forward slash, forward slash, insert here. Line 39, indented 4 times: n d, dot, set Left, left parenthesis, new, Integer Node, left parenthesis, value, right parenthesis, right parenthesis, semicolon. Line
40, indented 3 times: else, forward slash, forward slash, insert in left subtree. Line 41, indented 4 times: insert, left parenthesis, value, comma, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 42, indented twice: Right brace. Line 43, indented twice: else, forward slash, forward slash, insert somewhere right. Line 44, indented twice: Left brace. Line 45, indented 3 times: if, left parenthesis, n d, dot, get Right, left parenthesis, right parenthesis, equals, equals, null, right parenthesis, forward slash, forward slash, insert here. Line
46, indented 4 times: n d, dot, set Right, left parenthesis, new, Integer Node, left parenthesis, value, right parenthesis, right parenthesis, semicolon. Line 47, indented 3 times: else, forward slash, forward slash, insert in right subtree. Line 48, indented 4 times: insert, left parenthesis, value, comma, n d, dot, get Right, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 49, indented twice: Right brace. Line 50, indented once: Right brace. Line 51: Blank. Line 52,

indented once: Forward slash, asterisk, asterisk, search method. Line 53, indented once: asterisk, at, param, value an i n t value to search for in this B S T. Line 54, indented once: at, return true if value is found, comma, false otherwise. Line 55, indented once: asterisk, forward slash. Line 56, indented once: public, boolean, search, left parenthesis, i n t, value, right parenthesis. Line 57, indented once: Left brace. Line 58, indented twice: return, search, left parenthesis, value, comma, root, right parenthesis, semicolon. Line 59, indented once: Right brace. Line 60: Blank. Line 61, indented once: Forward slash, asterisk, asterisk, search recursive method. Line 62, indented once: asterisk, at, param, value an i n t to search for in the B S T "rooted" at n d. Line 63, indented once: asterisk, at, param n d an Integer Node reference. Line 64, indented once: asterisk, at, return a String represeboolean, comma, true if value was found. Line 65, indented once: asterisk in the B S T "rooted" at n d, comma, false otherwise. Line 66, indented once: Asterisk, forward slash. Line 67, indented once: private, boolean, search, left parenthesis, i n t, value, comma, Integer Node, n d, right parenthesis. Line 68, indented once: Left brace. Line 69, indented once: Left brace. Line 69, indented twice: if, left parenthesis, n d, equals, equals, null, right parenthesis, forward slash, forward slash, base case, comma, empty subtree. Line 70, indented 3 times: return, false, semicolon. Line 71, indented twice: else. Line 72, indented twice: Left brace. Line 73, indented 3 times: if, left parenthesis, n d, dot, get Data, left parenthesis, right parenthesis, equals, equals, value, right parenthesis, forward slash, forward slash, found value. Line 74, indented 4 times: return, true, semicolon. Line 75, indented 3 times: else, if, left parenthesis, n d, dot, get Data, left parenthesis, right parenthesis, greater than, value, right parenthesis, forward slash, forward slash, look left. Line 76, indented 4 times: return, search, left parenthesis, value, comma, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 77, indented 3 times: else, forward slash, forward slash, look right. Line 78, indented 4 times: return, search, left parenthesis, value, comma, n d, dot, get Right, left parenthesis, right parenthesis, semicolon. Line 79, indented twice: Right brace. Line 80, indented once: Right brace. Line 81: Blank. Line 82, indented once: Forward slash, asterisk, asterisk, max Value method. Line 83, indented once: Asterisk, at, param, n d an Integer Node reference, comma, not null. Line 84, indented once: Asterisk, at, return the largest value in the B S T "rooted" at n d. Line 85, indented once: Asterisk, forward slash. Line 86, indented once: private, i n t, max Value, left parenthesis, Integer Node, n d, right parenthesis. Line 87, indented once: Left brace. Line 88, indented twice: Integer Node, temp, equals, n d, semicolon. Line 89, indented twice: while, left parenthesis, temp, dot, get Right, left parenthesis, right parenthesis, exclamation mark, equals, null, right parenthesis. Line 90, indented 3 times: temp, equals, temp, dot, get Right, left parenthesis, right parenthesis, semicolon. Line 91, indented twice: return, temp, dot, get Data, left parenthesis, right parenthesis, semicolon. Line 92, indented once: Right brace. Line 93: Blank. Line 94, indented once: Forward slash, asterisk, asterisk, delete method. Line 95, indented once: Asterisk, at, param, value an i n t value to delete in this B S T. Line 96, indented once: Asterisk, forward slash. Line 97, indented once: public, void, delete, left parenthesis, i n t, value, right parenthesis. Line 98, indented once: Left brace. Line 99, indented twice: root, equals, delete, left parenthesis, value, comma, root, right parenthesis, semicolon. Line 100, indented once: Right brace. Line 101: Line 102, indented once: Forward slash, asterisk, asterisk, delete recursive method. Line 103, indented once: Asterisk, at, param, value an i n t to delete from in the B S T "rooted" at n d. Line 104, indented once: Asterisk, at, param, n d an Integer Node reference. Line 105, indented once: Asterisk, at, return an Integer Node. Line 106, indented once: Asterisk, forward slash. Line 107, indented once: private, Integer Node, delete, left parenthesis, i n t, value, comma, Integer Node, n d, right parenthesis. Line 108, indented once: Left brace. Line 109, indented twice: if, left parenthesis, n d, equals, equals, null, right parenthesis, forward slash, forward slash, base case, comma, empty subtree. Line 110, indented 3 times: return, null, semicolon. Line 111, indented twice: else, if, left parenthesis, n d, dot, get Data, left parenthesis, right parenthesis, equals, equals, value, right parenthesis, forward slash, forward slash, found value. Line 112, indented twice: Left brace. Line 113, indented 3 times: if, left parenthesis, n d, dot, get Right, left parenthesis, right parenthesis, equals, equals, null, right parenthesis, forward slash, forward slash, left child only or leaf. Line 114, indented 4 times: n d, equals, n d, dot, get Left, left parenthesis, right parenthesis, semicolon. Line 115, indented 3 times: else, if, left parenthesis, n d, dot, get Left, left parenthesis, right parenthesis, equals, equals, null, right parenthesis, forward slash, forward slash, right child only. Line 116, indented 4 times: n d, equals, n d, dot, get Right, left parenthesis, right parenthesis, semicolon. Line 117, indented 3 times: else, forward slash, forward slash, 2 children. Line 118, indented 3 times: Left brace. Line 119, indented 4 times: i n t, replacement, equals, max Value, left parenthesis, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 120, indented 4 times: n d, dot, set Data, left parenthesis, replacement, right parenthesis, semicolon. Line 121, indented 4 times: n d, dot, set Left, left parenthesis, delete, left parenthesis, replacement, comma, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 122, indented 3 times: Right brace. Line 123, indented twice: Right brace. Line 124, indented twice: else, if, left parenthesis, n d, dot, get Data, left parenthesis, right parenthesis, greater than, value, right parenthesis, forward slash, forward slash, delete in left subtree. Line 125, indented 3 times: n d, dot, set Left, left parenthesis, delete, left parenthesis, value, comma, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, right parenthesis, semicolon. Line 126, indented twice: else, forward slash, forward slash, delete in right subtree. Line 127, indented 3 times: n d, dot, set Right, left parenthesis, delete, left parenthesis, value, comma, n d, dot, get Right, left parenthesis, right parenthesis, right parenthesis, right parenthesis, semicolon. Line 128: Blank. Line 129, indented twice: return, n d, semicolon. Line 130, indented once: Right brace. Line 131: Blank. Line 132, indented once: Forward slash, asterisk, asterisk, rebalance method. Line 133, indented once: asterisk, at, param, a r r an array of i n t s sorted in ascending order. Line 134, indented once: Asterisk, rebuilds this B S T with the values in a r r. Line 135, indented once: asterisk, so that this B S T is now balanced. Line 136, indented once: Asterisk, forward slash. Line 137, indented once: public, void, rebalance, left parenthesis, i n t, left square brace, right square brace, a r r, right parenthesis. Line 138, indented once: Left brace. Line 139, indented twice: root, equals, rebalance, left parenthesis, a r r, comma, 0, comma, a r r, dot, length, minus, 1, right parenthesis, semicolon. Line 140, indented once: Right brace. Line 141: Blank. Line 142, indented once: Forward slash, asterisk, asterisk, rebalance recursive method. Line 143, indented once: Asterisk, at, param, a r r an array of i n t s sorted in ascending order. Line 144, indented once: Asterisk, at, param, start an i n t, comma, a valid index of a r r. Line 145, indented once: Asterisk, at, param end an i n t, comma, a valid index of a r r. Line 146, indented once:

Asterisk, at, return an Integer Node, comma, the "root" of a balanced B S T. Line 147, indented once: Asterisk, made up of all the elements of a r r between start and end. Line 148, indented once: Asterisk, forward slash. Line 149, indented once: private, Integer Node, rebalance, left parenthesis, i n t, left square brace, right square brace, a r r, comma, i n t, start, comma, i n t, end, right parenthesis. Line 150, indented once: Left brace. Line 151, indented twice: if, left parenthesis, start, less than or equals, end, right parenthesis. Line 152, indented twice: Left brace. Line 153, indented 3 times: i n t, middle, equals, left parenthesis, start, plus, end, right parenthesis, forward slash, 2, semicolon. Line 154, indented 3 times: Integer Node, n d, equals, new, Integer Node, left parenthesis, a r r, left square brace, middle, right square brace, right parenthesis, semicolon. Line 155, indented 3 times: n d, dot, set Left, left parenthesis, rebalance, left parenthesis, a r r, comma, start, comma, middle, minus, 1, right parenthesis, right parenthesis, semicolon. Line 156, indented 3 times: n d, dot, set Right, left parenthesis, rebalance, left parenthesis, a r r, comma, middle, plus, 1, comma, end, right parenthesis, right parenthesis, semicolon. Line 157, indented 3 times: return, n d, semicolon. Line 158, indented twice: Right brace. Line 159, indented twice: else. Line 160, indented twice: return, null, semicolon. Line 161, indented once: Right brace. Line 162: Blank. Line 163, indented once: Forward slash, asterisk, asterisk, pre Order method. Line 164, indented once: Asterisk, at, return a String representation of the pre order. Line 165, indented once: Asterisk, traversal of this Binary Tree. Line 166, indented once: Asterisk, forward slash. Line 167, indented once: public, String, pre Order, left parenthesis, right parenthesis. Line 168, indented once: Left brace. Line 169, indented twice: return, pre Order, left parenthesis, root, right parenthesis, semicolon. Line 170, indented once: Right brace. Line 171: Blank. Line 172, indented once: Forward slash, asterisk, asterisk, pre Order recursive method. Line 173, indented once: Asterisk, at, param, n d an Integer Node reference. Line 174, indented once: Asterisk, at, return a String representation of the pre order. Line 175, indented once: Asterisk, traversal of the Binary Tree "rooted" at n d. Line 176, indented once: Asterisk, forward slash. Line 177, indented once: private, String, pre Order, left parenthesis, Integer Node, n d, right parenthesis. Line 178, indented once: Left brace. Line 179, indented twice: if, left parenthesis, n d, not equals, null, right parenthesis. Line 180, indented twice: Left brace. Line 181, indented 3 times: Forward slash, forward slash, visit n d. Line 182, indented 3 times: String, result, equals, n d, dot, get Data, left parenthesis, right parenthesis, plus, open quotes, space, close quotes, semicolon. Line 183, indented 3 times: Forward slash, forward slash, visit left subtree in pre order. Line 184, indented once: result, plus, equals, pre Order, left parenthesis, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 185, indented 3 times: Forward slash, forward slash, visit right subtree in pre order. Line 186, indented 3 times: result, plus, equals, pre Order, left parenthesis, n d, dot, get Right, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 187, indented 3 times: return, result, semicolon. Line 188, indented twice: Right brace. Line 189, indented twice: else. Line 190, indented 3 times: return, open quotes, close quotes, semicolon. Line 191, indented once: Right brace. Line 192: Blank. Line 193, indented once: Forward slash, asterisk, asterisk, in Order method. Line 194, indented once: Asterisk, at, return, a String representation of the in order. Line 195, indented once: asterisk, traversal of this Binary Tree. Line 196, indented once: Asterisk, forward slash. Line 197, indented once: public, String, in Order, left parenthesis, right parenthesis. Line 198, indented once: Left brace. Line 199, indented twice: return, in Order, left parenthesis, root, right parenthesis, semicolon. Line 200, indented once: Right brace. Line 201: Blank. Line 202, indented once: Forward slash, asterisk, asterisk, in Order recursive method. Line 203, indented once: Asterisk, at, param n d an Integer Node reference. Line 204, indented once: Asterisk, at, return, a String representation of the in order. Line 205, indented once: Asterisk, traversal of the Binary Tree "rooted" at n d. Line 206, indented once: Asterisk, forward slash. Line 207, indented once: private, String, in Order, left parenthesis, Integer Node, n d, right parenthesis. Line 208, indented once: Left brace. Line 209, indented twice: if, left parenthesis, n d, not equals, null, right parenthesis. Line 210, indented twice: Left brace. Line 211, indented 3 times: Forward slash, forward slash, visit left subtree in order. Line 212, indented 3 times: String, result, equals, in Order, left parenthesis, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 213, indented 3 times: Forward slash, forward slash, visit n d. Line 214, indented 3 times: result, plus, equals, n d, dot, get Data, left parenthesis, right parenthesis, plus, open quotes, space, close quotes, semicolon. Line 215, indented 3 times: Forward slash, forward slash, visit right subtree in order. Line 216, indented 3 times: result, plus, equals, in Order, left parenthesis, n d, dot, get Right, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 217, indented 3 times: return, result, semicolon. Line 218, indented twice: Right brace. Line 219, indented twice: else. Line 220, indented 3 times: return, open quotes, close quotes, semicolon. Line 221, indented once: Right brace. Line 222: Blank. Line 223, indented once: Forward slash, asterisk, asterisk, post Order method. Line 224, indented once: Asterisk, at, return, a String representation of the post order. Line 225, indented once: Asterisk, traversal of this Binary Tree. Line 226, indented once: Asterisk, forward slash. Line 227, indented once: public, String, post Order, left parenthesis, right parenthesis. Line 228, indented once: Left brace. Line 229, indented twice: return, post Order, left parenthesis, root, right parenthesis, semicolon. Line 230, indented once: Right brace. Line 231: Blank. Line 232, indented once: Forward slash, asterisk, asterisk, post Order recursive method. Line 233, indented once:

asterisk, at, param, n d, an Integer Node reference. Line 234, indented once: Asterisk, at, return, a String representation of the post order. Line 235, indented once: asterisk, traversal of the Binary Tree "rooted" at n d. Line 236, indented once: Asterisk, forward slash. Line 237, indented once: private, String, post Order, left parenthesis, Integer Node, n d, right parenthesis. Line 238, indented once: Left brace. Line 239, indented twice: if, left parenthesis, n d, not equals, null, right parenthesis. Line 240, indented twice: Left brace. Line 241, indented 3 times: Forward slash, forward slash, visit left subtree in order. Line 242, indented 3 times: String, result, equals, post Order, left parenthesis, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 243, indented 3 times: Forward slash, forward slash, visit right subtree in order. Line 244, indented 3 times: result, plus, equals, post Order, left parenthesis, n d, dot, get Right, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 245, indented 3 times: Forward slash, forward slash, visit n d. Line 246, indented 3 times: result, plus, equals, n d, dot, get Data, left parenthesis, right parenthesis, plus, open quotes, plus, close quotes, semicolon. Line 247, indented 3 times: return, result, semicolon. Line 248, indented twice: Right brace. Line 249, indented twice: else. Line 250, indented 3 times: return, open quotes, close quotes, semicolon. Line 251, indented once: Right brace. Line 252, right brace.

Back to Example

A table detailing the recursive calls of the delete method when deleting 5. The node has one child. The table has 3 rows and 5 columns. It has the following column headings. Method call. Argument (n d). Returned node. Setting children. Line. The row details in the table are the following. Row 1: Method call, delete, left square brace, 5, comma, root, right square brace; Argument (n d), node, left parenthesis, 50, right parenthesis; Returned node, node, left square brace, 50, right square brace; Setting children, node, left parenthesis, 50, right parenthesis, set Left, left, left square brace, node, left parenthesis, 20, right parenthesis, right square brace; Line, 125. Row 2: Method call, delete, left square brace, 5, comma, node, left square brace, 20, right square brace, right square brace; Argument (n d), node, left parenthesis, 20, right parenthesis; Returned node, node, left parenthesis, 20, right parenthesis; Setting children, node, left parenthesis, 20, right parenthesis, set Left, left, left square brace, node, left parenthesis, 17, right parenthesis, right square brace; Line, 125. Row 3: Method call, delete, left square brace, 5, comma, node, left square brace, 5, right square brace, right square brace; Argument (n d), node, left square brace, 5, right square brace; Returned node, node, left square brace, 17, right square brace; Setting children, Blank; Line, Blank.

Back to Table

Code for the Binary Search Tree Client class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, The Binary Search Tree Client class. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: public, class, Binary Search Tree Client. Line 6: Left brace. Line 7, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 8, indented once: Left brace. Line 9, indented twice: Binary Search Tree, b s t, equals, new, Binary Search Tree, left parenthesis, right parenthesis, semicolon. Line 10, indented twice: i n t, left square brace, right square brace, insertions, equals, left brace, 50, comma, 20, comma, 36, comma, 66, comma, 25, comma, 5, comma, 17, comma, 84, comma, 44, right brace, semicolon. Line 11, indented twice: for, left parenthesis, i n t, value, colon, insertions, right parenthesis. Line 12, , indented 3 times: b s t, dot, insert, left parenthesis, value, right parenthesis, semicolon. Line 13: Blank. Line 14, indented twice: System, dot, out, dot, print l n, left parenthesis, open quotes, Pre order, colon, close quotes, plus, b s t, dot, pre Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 15, indented twice: System, dot, out, dot, print l n, left parenthesis, open quotes, In order, colon, close quotes, plus, b s t, dot, in Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 16, indented twice: System, dot, out, dot, print l n, left parenthesis, open quotes, Post order, colon, close quotes, plus, b s t, dot, post Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 17: Blank. Line 18, indented twice: i n t, left square brace, right square brace, deletions, equals, left brace, 50, comma, 20, comma, 36, comma, 66, comma, 9, comma, 84, comma, 17, comma, 25, comma, 5, comma, 44, comma, 8, right brace, semicolon. Line 19, indented twice: for, left parenthesis, i n t, value, colon, deletions, right parenthesis. Line 20, indented twice: Left brace. Line 21, indented 3 times: System, dot, out, dot, print l n, left parenthesis, open quotes, back slash, n, Deleting, close quotes, plus, value, right parenthesis, semicolon. Line 22, indented 3 times: b s t, dot, delete, left parenthesis, value, right parenthesis, semicolon. Line 23, indented 3 times. System, dot, out, dot, print l n, left parenthesis, open quotes, Pre order, colon, close quotes, plus, b s t, dot, pre Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 24, indented 3 times. System, dot, out, dot, print l n, left parenthesis, open quotes, In order, colon, close quotes, plus, b s t, dot, in Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 25, indented 3 times: System, dot, out, dot, print l n, left parenthesis, open quotes, Post order, colon, close quotes, plus, b s t, dot, post Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 26, indented twice: Right brace. Line 27, indented once: Right brace. Line 28: Right brace.

Back to Example

Code for the Binary Search Tree Client 2 class. Program code. In the code, the words in the variable names are merged. Code for the Binary Search Tree Client 2 class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, The Binary Search Tree Client 2 class. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: public, class, Binary Search Tree Client 2. Line 6: Left brace. Line 7, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 8, indented once: Left brace. Line 9, indented twice: Binary Search Tree, b s t, equals, new, Binary Search Tree, left parenthesis, right parenthesis, semicolon. Line 10, indented twice: i n t, left square brace, right square brace, insertions, equals, left brace, 5, comma, 17, comma, 20, comma, 25, comma, 36, comma, 44, comma, 50, comma, 66, comma, 84, right brace, semicolon. Line 11, indented twice: for, left parenthesis, i n t, value, colon, insertions, right parenthesis. Line 12, indented 3 times: b s t, dot, insert, left parenthesis, value, right parenthesis, semicolon. Line 13: Blank. Line 14, indented twice: System, dot, out, dot, print l n, left parenthesis, open quotes, Pre order, colon, space, close quotes, plus, b s t, dot, pre Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 15, indented twice: System, dot, out, dot, print l n, left parenthesis, open quotes, In order, colon, space, close quotes, plus, b s t, dot, in Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 16, indented twice: System, dot, out, dot, print l n, left parenthesis, open quotes, Post order, colon, space, close quotes, plus, b s t, dot, post Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 17: Blank. Line 18, indented twice: b s t, dot, rebalance, left parenthesis, insertions, right parenthesis, semicolon. Line 19: Blank. Line 20, indented twice: System, dot, out, dot, print l n, left parenthesis, open quotes, back slash, n, After rebuilding the tree, close quotes, right parenthesis, semicolon. Line 21, indented twice: System, dot, out, dot, print l n, left parenthesis, open quotes, Pre order, colon, space, close quotes, plus, b s t dot pre Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 22, indented twice: System, dot, out, dot, print l n, left parenthesis, open quotes, In order, colon, space, close quotes, plus, b s t, dot, in Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 23, indented twice: System, dot, out, dot, print l n, left parenthesis, open quotes, Post order, colon, space, close quotes, plus, b s t dot, post Order, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 24, indented once: Right brace. Line 25: Right brace.

Back to Example

Tables detailing constructors and methods of the J Frame class. Table detailing constructors of the J Frame class. The table has 2 rows and 2 columns. It has the following column headings. Class. Constructor. The row details in the table are the following. Row 1: Class, J Frame; Constructor, J Frame, left parenthesis, right parenthesis, constructs a J Frame object, initially invisible, with no text in the title bar. Row 2: Class, J Frame; Constructor, J Frame, left parenthesis, String, title Bar Text, right parenthesis constructs a J Frame object, initially invisible, with title Bar Text displayed in the window's title bar. Table detailing methods of the J Frame class. The table has 4 rows and 2 columns. It has the following column headings. Return Value. Method name and argument list. The row details in the table are the following. Row 1: Return value, Container; Method name and argument list, get Content Pane, left parenthesis, right parenthesis returns the content pane object for this window. Row 2: Return value, void; Method name and argument list, set Default Close Operation, left parenthesis, i n t, operation, right parenthesis sets the default operation when the user closes this window, that is, when the user clicks on the X icon in the top-right corner of the window. Row 3: Return value, void; Method name and argument list, set Size, left parenthesis, i n t, width, i n t, height, right parenthesis sizes the window to the specified width and height in pixels. Row 4: Return value, void; Method name and argument list, set Visible, left parenthesis, boolean, mode, right parenthesis displays this window is mode is true, hides the window if mode is false.

Back to Table

Code for a shell G U I application using J Frame. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, A Shell G U I Application. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x, dot, swing, dot, J Frame, semicolon. Line 6: import, java, dot, a w t, dot, Container, semicolon. Line 7, forward slash, forward slash, other import statements here as needed. Line 8: Blank. Line 9: public, class, Shell G U I Application, extends, J Frame. Line 10: Left brace. Line 11, indented once: private, Container, contents, semicolon. Line 12, indented once: Forward slash, forward slash, declare other instance variables. Line 13: Blank. Line 14, indented once: Forward slash, forward slash, constructor. Line 15, indented once: public, Shell G U I Application, left parenthesis, right parenthesis. Line 16, indented once: Left brace. Line 17, indented twice: Forward slash, forward slash, call J Frame constructor with title bar text. Line 18, indented twice: super, left parenthesis, open quotes, A Shell G U I Application, close quotes, right parenthesis, semicolon. Line 19: Blank. Line 20, indented twice: Forward slash, forward slash, get container for components. Line 21, indented twice: contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 22: Blank. Line 23, indented twice: Forward slash, forward slash, set the layout manager. Line 24: Blank. Line 25, indented twice:

Forward slash, forward slash, instantiate G U I components and other instance variables. Line 26: Blank. Line 27, indented twice: Forward slash, forward slash, add G U I components to the content pane. Line 28: Blank. Line 29, indented twice: Forward slash, forward slash, set original size of window. Line 30, indented twice: set Size, left parenthesis, 300, comma, 200, right parenthesis, semicolon. Line 31: Blank. Line 32, indented twice: Forward slash, forward slash, make window visible. Line 33, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 34, indented once: Right brace. Line 35: Blank. Line 36, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 37, indented once: Left brace. Line 38, indented twice: Shell G U I Application basic G u i, equals, new, Shell G U I Application, left parenthesis, right parenthesis, semicolon. Line 39, indented twice: basic G u i, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 40, indented once: Right brace. Line 41: Right brace.

Back to Example

A table with details of selected G U I components and Java classes. The table has 8 rows and 3 columns. It has the following column headings. Component. Purpose. Java Class. The row details in the table are the following. Row 1: Component, Label; Purpose, Displays an image or read-only text. Labels are often paired with text fields and are used to identify the contents of the text field; Java Class, J Label. Row 2: Component, Text field; Purpose, A single-line text box for displaying information and accepting user input.; Java Class, J Text Field. Row 3: Component, Text area; Purpose, Multiple-line text field for data entry or display; Java Class, J Text Area. Row 4: Component, Password field; Purpose, Single-line text field for accepting passwords without displaying the characters typed; Java Class, J Password Field. Row 5: Component, Button; Purpose, Command button that the user clicks to signal that an operation should be performed; Java Class, J Button. Row 6: Component, Radio button; Purpose, Toggle button that the user clicks to select one option in the group. Clicking on a radio button deselects any previously selected option; Java Class, J Radio Button. Row 7: Component, Checkbox; Purpose, Toggle button that the user clicks to select 0, 1, or more options in a group; Java Class, J Check Box. Row 8: Component, Drop-down list; Purpose, Drop-down list of items that the user clicks to select one item; Java Class, J Combo Box.

Back to Table

A table detailing some useful J Component methods. Methods of the J Component class. The table has 6 rows and 2 columns. It has the following column headings. Return Value. Method name and argument list. The row details in the table are the following. Row 1: Return value, void; Method name and argument list, set Visible, left parenthesis, boolean, mode, right parenthesis, makes the component visible if mode is true; hides the component if mode is false. The default is visible. Row 2: Return value, void; Method name and argument list, set Tool Tip Text, left parenthesis, String, tool Tip, right parenthesis, sets the tool tip text to tool Tip. When the mouse lingers over the component, the tool tip text will be displayed. Row 3: Return value, void; Method name and argument list, set Foreground, left parenthesis, Color, fore Color, right parenthesis, sets the foreground color of the component to fore Color. Row 4: Return value, void; Method name and argument list, set Background, left parenthesis, Color, back Color, right parenthesis, sets the background color of the component to back Color. Row 5: Return value, void; Method name and argument list, set Opaque, left parenthesis, boolean, mode, right parenthesis, sets the component's background to opaque if mode is true; sets the component's background to transparent if mode is false. If opaque, the component's background is filled with the component's background color; if transparent, the component's background is filled with the background color of the container on which it is placed. The default is transparent. Row 6: Return value, void; Method name and argument list, set Enabled, left parenthesis, boolean, mode, right parenthesis, enables the component if mode is true; disables the component if mode is false. An enabled component can respond to user input.

Back to Table

Code for a program that uses a simple G U I component, J Label. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Using J Labels to display text and images. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import java dot a w t dot Container, semicolon. Line 6: import, java x dot swing dot J Frame, semicolon. Line 7: import, java x dot swing dot Image Icon, semicolon. Line 8: import, java x dot swing dot J Label, semicolon. Line 9: import, java dot a w t dot Flow Layout, semicolon. Line 10: import, java dot a w t dot Color, semicolon. Line 11: Blank. Line 12: public, class, Dinner, extends, J Frame. Line 13: Left brace. Line 14, indented once: private, Container, contents, semicolon. line 15, indented once: private, J Label, label Text, semicolon. Line 16, indented once: private, J Label, label Image, semicolon. Line 17: Blank. Line 18, indented once: Forward slash, forward slash, Constructor. Line 19, indented once: public, Dinner, left parenthesis, right parenthesis. Line 20, indented once: Left brace. Line 21, indented twice: super, left parenthesis, open quotes, What’s for dinner?, close quotes, right parenthesis, semicolon, tab space, forward slash, forward slash, call J Frame constructor. Line 22: Blank. Line 23, indented twice: contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon, tab space, forward slash, forward slash, get content pane. Line 24: Blank. Line 25, indented twice: contents dot set Layout, left parenthesis, new, Flow Layout, left parenthesis, right

parenthesis, right parenthesis, semicolon, tab space, forward slash, forward slash, set layout manager. Line 26: Blank. Line 27, indented twice: Forward slash, forward slash, use the J Label constructor with a String argument. Line 28, indented twice: label Text, equals, new, J Label, left parenthesis, open quotes, Sushi tonight?, close quotes, right parenthesis, semicolon. Line 29: Blank. Line 30, indented twice: forward slash, forward slash, set label properties. Line 31, indented twice: label Text dot set Foreground, left parenthesis, Color dot WHITE, right parenthesis, semicolon. Line
32, indented twice: label Text dot set Background, left parenthesis, Color dot BLUE, right parenthesis, semicolon. Line 33, indented twice: label Text dot set Opaque, left parenthesis, true, right parenthesis, semicolon. Line 34: Blank. Line 35, indented twice: Forward slash, forward slash, use the J Label constructor with an Image Icon argument. Line 36, indented twice: label Image, equals, new, J Label, left parenthesis, new, Image Icon, left parenthesis, open quotes, sushi dot j p g, close quotes, right parenthesis, right parenthesis, semicolon. Line 37: Blank. Line 38, indented twice: Forward slash, forward slash, set tool tip text. Line 39, indented twice: label Image dot set Tool Tip Text, left parenthesis, open quotes, photo of sushi, close quotes, right parenthesis, semicolon. Line 40: Blank. Line 41, indented twice: Forward slash, forward slash, add the two labels to the content pane. Line 42, indented twice: contents dot add, left parenthesis, label Text, right

parenthesis, semicolon. Line 43, indented twice: contents dot add, left parenthesis, label Image, right parenthesis, semicolon. Line 44: Blank. Line 45, indented twice: set Size, left parenthesis, 300, comma, 200, right parenthesis, semicolon. Line 46, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 47, indented once: Right brace. Line 48: Blank. Line 49, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 50, indented once: Left brace. Line 51, indented twice: Dinner, dinner, equals, new, Dinner, left parenthesis, right parenthesis, semicolon. Line 52, indented twice: dinner dot set Default Close Operation, left parenthesis, J Frame dot EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 53, indented once: Right brace. Line 54: Right brace.

Back to Example

A table detailing useful methods of the Container class. Methods of the Container class. The table has 3 rows and 2 columns. It has the following column headings. Return Value. Method name and argument list. The row details in the table are the following. Row 1: Return value, void; Method name and argument list, set Layout, left parenthesis, Layout Manager, m g r, right parenthesis, sets the layout manager of the window to m g r. Row 2: Return value, Component; Method name and argument list, add, left parenthesis, Component, component, right parenthesis, adds the component to the container, using the rules of the layout manager. Returns the argument. Row 3: Return value, void; Method name and argument list, remove All, left parenthesis, right parenthesis, removes all components from the window.

Back to Table

A table detailing useful constructors of the J Label Class. Constructors of the J Label Class. The table has 3 rows and 2 columns. It has the following column headings. Class. Constructor. The row details in the table are the following. Row 1: Class, J Label; Constructor, J Label, left parenthesis, String, text, right parenthesis, creates a J Label object that displays the specified text. Row 2: Class, J Label; Constructor, J Label, left parenthesis, String, text, comma, i n t, alignment, right parenthesis, creates a J Label object that displays the specified text. The alignment argument specifies the alignment of the text within the label component. The alignment value can be any of the following static i n t constants from the Swing Constants interface: LEFT, CENTER, RIGHT, LEADING, or TRAILING. By default, the label text is left-adjusted. Row 3: Class, J Label; Constructor, J Label, left parenthesis, Icon, image, right parenthesis, creates a J Label object that displays the image.

Back to Table

Table detailing the get Source method inherited by Event classes. The return value is Object. The method name and argument list are the following. get Source, left parenthesis, right parenthesis, returns the object reference of the component that fired the event.

Back to Table

A table detailing component events and listeners. The table has 8 rows and 4 columns. It has the following column headings. J Component. User Interaction That Generates an Event. The Event Object Created. Listener Interface that Event Handler Should Implement. The row details in the table are the following. Row 1: J Component, J Text Field; User Interaction That Generates an Event, Pressing Enter; The Event Object Created, Action Event; Listener Interface that Event Handler Should Implement, Action Listener. Row 2: J Component, J Button; User Interaction That Generates an Event, Pressing the button; The Event Object Created, Action Event; Listener Interface that Event Handler Should Implement, Action Listener. Row 3: J Component, J Radio Button; User Interaction That Generates an Event, Selecting a radio button; The Event Object Created, Item Event; Listener Interface that Event Handler Should Implement, Item Listener. Row 4: J Component, J Check Box; User Interaction That Generates an Event, Selecting or deselecting a checkbox; The Event Object Created, Item Event; Listener Interface that Event Handler Should Implement, Item Listener. Row 5: J Component, J List; User Interaction That Generates an Event, Selecting an item; The Event Object Created, List Selection Event; Listener Interface that Event Handler Should Implement, List Selection Listener. Row 6: J Component, J Combo Box; User Interaction That Generates an Event, Selecting an item; The Event Object Created, Item Event; Listener Interface that Event Handler Should Implement, Item Listener. Row 7: J Component, Any component; User Interaction That Generates an Event, Pressing or releasing mouse buttons; The Event Object Created, Mouse Event; Listener Interface that Event Handler Should Implement, Mouse Listener. Row 8: J Component, Any component; User Interaction That Generates an Event, Moving or dragging the mouse; The Event Object Created, Mouse Event; Listener Interface that Event Handler Should Implement, Mouse Motion Listener.

Back to Table

A table detailing some add Listener methods. Add Listener Method A P Is. The table has 5 rows. The row details in the table are the following. Row 1: void, add Action Listener, left parenthesis, Action Listener, handler, right parenthesis. Row 2: void, add Item Listener, left parenthesis, Item Listener, handler, right parenthesis. Row 3: void add List Selection Listener, left parenthesis, List Selection Listener, handler, right parenthesis. Row 4: void, add Mouse Listener, left parenthesis, Mouse Listener, handler, right parenthesis. Row 5: void, add Mouse Motion Listener, left parenthesis, Mouse Motion Listener, handler, right parenthesis.

Back to Table

A table detailing useful constructors and methods of J Text Field, J Text Area, and J Password Field. Constructors and Methods of the J Text Field, J Text Area, and J Password Field classes. Constructors. The table has 6 rows and 2 columns. Class. Constructor. The row details in the table are the following. Row 1: Class, J Text Field; Constructor, J Text Field, left parenthesis, String, text, comma, i n t, num Columns, right parenthesis, constructs a text field initially filled with text, with the specified number of columns. Row 2: Class, J Text Field; Constructor, J Text Field, left parenthesis, i n t, number Columns, right parenthesis, constructs an empty text field with the specified number of columns. Row 3: Class, J Text Area; Constructor, J Text Area, left parenthesis, String, text, right parenthesis, constructs a text area initially filled with text. Row 4: Class, J Text Area; Constructor, J Text Area, left parenthesis, i n t, num Rows, comma, i n t, num Columns, right parenthesis, constructs an empty text area with the number of rows and columns specified by num Rows and num Columns. Row 5: Class, J Text Area; Constructor, J Text Area, left parenthesis, String, text, comma, i n t, num Rows, comma, i n t, num Columns, constructs a text area initially filled with text, and with the number of rows and columns specified by num Rows and num Columns. Row 6: Class, J Password Field; Constructor, J Password Field, left parenthesis, i n t, number Columns, right parenthesis, constructs an empty password field with the specified number of columns. Methods Common to J Text Field, J Text Area, and J Password Field Classes. The table has 3 rows and 2 columns. It has the following column headings. Return Value. Method name and argument list. The row details in the table are the following. Row 1: Return value, void; Method name and argument list, set Editable, left parenthesis, boolean, mode, right parenthesis, sets the properties of the text component as editable or noneditable, depending on whether mode is true or false. The default is editable. Row 2: Return value, void; Method name and argument list, set Text, left parenthesis, String, new Text, right parenthesis, sets the text of the text component to new Text. Row 3: Return value, String; Method name and argument list, get Text, left parenthesis, right parenthesis, returns the text contained in the text component. Additional methods of the J Password Field class. The table has 2 rows and 2 columns, It has the following column headings. Return value. Method name and argument list. The row details in the table are the following. Row 1: Return value, void; Method name and argument list, set Echo Char, left parenthesis, char, c, right parenthesis. Row 2: Return value, char, left square brace, right square brace; Method name and argument list, get Password, left parenthesis, right parenthesis, returns the text entered in this password field as an array of chars. Note: This method is preferred over the get Text method for getting the password typed by the user.

Back to Table

Code for carrying out simple Math using J Button. Program code. In the code, the words in the variable names are merged. 1 Forward slash, asterisk, Simple Math Operations Using J Buttons. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x dot swing dot asterisk, semicolon. Line 6: import java dot a w t dot asterisk, semicolon. Line 7: import, java dot a w t dot event dot asterisk, semicolon. Line 8: Blank. Line 9: public, class, Simple Math, extends, J Frame. Line 10: Left brace. Line 11, indented once: private, Container, contents, semicolon. Line 12, indented once: private, J Label, operand Label, comma, result Label, comma, result, semicolon. Line 13, indented once: private, J Text Field, operand, semicolon. Line 14, indented once: private, J Button, square, comma, cube, semicolon. Line 15: Blank. Line 16, indented once: public Simple Math, left parenthesis, right parenthesis. Line 17, indented once: Left brace. Line 18, indented twice: super, left parenthesis, open quotes, Simple Math, close quotes, right parenthesis, semicolon. Line 19, indented twice: contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 20, indented twice: contents, dot, set Layout, left parenthesis, new, Flow Layout, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 21: Blank. Line 22, indented twice: operand Label, equals, new, J Label, left parenthesis, open quotes, Enter a number, close quotes, right parenthesis, semicolon, forward slash, forward slash, text field label. Line 23, indented twice: operand, equals, new, J Text Field, left parenthesis, 5, right parenthesis, semicolon, forward slash, forward slash, text field is 5 characters wide. Line 24: Blank. Line 25, indented twice: forward slash, forward slash, instantiate buttons. Line 26, indented twice: square, equals, new, J Button, left parenthesis, open quotes, Square, close quotes, right parenthesis, semicolon. Line 27, indented twice: cube, equals, new, J Button, left parenthesis, open quotes, Cube, close quotes, right parenthesis, semicolon. Line 28: Blank. Line 29, indented twice: result Label, equals, new, J Label, left parenthesis, open quotes, Result, colon, close quotes, right parenthesis, semicolon, forward slash, forward slash, label for result. Line 30, indented twice: result, equals, new, J Label, left parenthesis, open quotes, question mark, question mark, question mark, close quotes, right parenthesis, semicolon, forward slash, forward slash, label to hold result. Line 31: Blank. Line 32, indented twice: forward slash, forward slash, add components to the window. Line 33, indented twice: contents, dot, add, left parenthesis, operand Label, right parenthesis, semicolon. Line 34, indented twice: contents, dot, add, left parenthesis, operand, right parenthesis, semicolon. Line 35, indented twice: contents, dot, add, left parenthesis, square, right parenthesis, semicolon. Line 36, indented twice: contents, dot, add, left parenthesis, cube, right parenthesis, semicolon. Line 37, indented twice: contents, dot, add, left parenthesis, result Label, right parenthesis, semicolon. Line 38, indented twice: contents, dot, add, left parenthesis, result, right parenthesis, semicolon. Line 39: Blank. Line 40, indented twice: forward slash, forward slash, instantiate our event handler. Line 41, indented twice: Button Handler, b h, equals, new, Button Handler, left parenthesis, right parenthesis, semicolon. Line 42: Blank. Line 43, indented twice: Forward slash, forward slash, add event handler as listener for both buttons. Line 44, indented twice: square, dot, add Action Listener, left parenthesis, b h, right parenthesis, semicolon. Line 45, indented twice: cube, dot, add Action Listener, left parenthesis, b h, right parenthesis, semicolon. Line 46: Blank. Line 47, indented twice: set Size, left parenthesis, 175, comma, 150, right parenthesis, semicolon. Line 48, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 49, indented once: Right brace. Line 50: Blank. Line 51, indented once. forward slash, forward slash, private inner class event handler. Line 52, indented once. private, class, Button Handler, implements, Action Listener. Line 53, indented once: left brace. Line 54, indented twice: forward slash, forward slash, implement action Performed method. Line 55, indented twice: public, void, action Performed, left parenthesis, Action Event, a e, right parenthesis. Line 56, indented twice: Left brace. Line 57, indented 3 times: try. Line 58, indented 3 times: Left brace. Line 59, indented 4 times: double, o p, equals, Double, dot, parse Double, left parenthesis, operand, dot, get Text, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 60: Blank. Line 61, indented 4 times: forward slash, forward slash, identify which button was pressed. Line 62, indented 4 times: if, left parenthesis, a e, dot, get Source, left parenthesis, right parenthesis, equals, equals, square, right parenthesis. Line 63, indented 5 times: result, dot, set Text, left parenthesis, Double, dot, value Of, left parenthesis, o p, asterisk, o p, right parenthesis, dot, to String, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 64, indented 4 times: else, if, left parenthesis, a e, dot, get Source, left parenthesis, right parenthesis, equals, equals, cube, right parenthesis. Line 65, indented 5 times: result, dot, set Text, left parenthesis, Double, dot, value Of, left parenthesis, o p, asterisk, o p, asterisk, o p, right parenthesis, dot, to String, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 66, indented 3 times: Right brace. Line 67, indented 3 times: catch, left parenthesis, Number Format Exception, e, right parenthesis. Line 68, indented 3 times: Left brace. Line 69, indented 4 times: operand Label, dot, set Text, left parenthesis, open quotes, Enter a number, close quotes, right parenthesis, semicolon. Line 70, indented 4 times: operand, dot, set Text, left parenthesis, open quotes, close quotes, right parenthesis, semicolon. Line 71, indented 4 times: result, dot, set Text, left parenthesis, open quotes, question mark, question mark, question mark, question mark, close quotes, right parenthesis, semicolon. Line 72, indented 3 times: Right brace. Line

73, indented twice: Right brace. Line 74, indented once: Right brace. Line 75: Blank. Line 76, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 77, indented once: Left brace. Line 78, indented twice: Simple Math, s m, equals, new, Simple Math, left parenthesis, right parenthesis, semicolon. Line 79, indented once: s m, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 80, indented once: Right brace. Line 81: Right brace.

Back to Example

Program code. In the code, the words in the variable names are merged. Line 1: public, void, action Performed, left parenthesis, Action Event, event, right parenthesis.

Back to Table

Code for a program that does simple math using J Button. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Simple Math Operations Using J Buttons. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x dot swing dot asterisk, semicolon. Line 6: import, java dot a w t dot asterisk, semicolon. Line 7: import, java dot a w t dot event dot asterisk, semicolon. Line 8: Blank. Line 9: public, class, Simple Math, extends, J Frame. Line 10:

Left brace. Line 11, indented once: private, Container, contents, semicolon. Line 12, indented once: private, J Label, operand Label, comma, result Label, comma, result, semicolon. Line 13, indented once: private, J Text Field, operand, semicolon. Line 14, indented once: private, J Button, square, comma, cube, semicolon. Line 15: Blank. Line 16, indented once: public, Simple Math, left parenthesis, right parenthesis. Line 17, indented once: Left brace. Line 18, indented twice: super, left parenthesis, open quotes, Simple Math, close quotes, right parenthesis, semicolon. Line 19, indented twice: contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 20, indented twice: contents, dot, set Layout, left parenthesis, new, Flow Layout, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 21: Blank. Line 22, indented twice: operand Label, equals, new, J Label, left parenthesis, open quotes, Enter a number, close quotes, right parenthesis, semicolon, forward slash, forward slash, text field label. Line 23, indented twice: operand, equals, new, J Text Field, left parenthesis, 5, right parenthesis, semicolon, forward slash, forward slash, semicolon, forward slash, forward slash, text field is 5 characters wide. Line 24: Blank. Line 25, indented twice: forward slash, forward slash, instantiate buttons. Line 26, indented twice: square, equals, new, J Button, left parenthesis, open quotes, Square, close quotes, right parenthesis, semicolon. Line 27, indented twice: cube, equals, new, J Button, left parenthesis, open quotes, Cube, close quotes, right parenthesis, semicolon. Line 28: Blank. Line 29, indented twice: result Label, equals, new, J Label, left parenthesis, open quotes, Result, colon, close quotes, right parenthesis, semicolon, forward slash, forward slash, label for result. Line 30, indented twice: result, equals, new, J Label, left parenthesis, open quotes, question mark, question mark, question mark, close quotes, right parenthesis, semicolon, forward slash, forward slash, label to hold result. Line 31: Blank. Line 32, indented twice: forward slash, forward slash, add components to the window. Line 33, indented twice: contents, dot, add, left parenthesis, operand Label, right parenthesis, semicolon. Line 34, indented twice: contents, dot, add, left parenthesis, operand, right parenthesis, semicolon. Line 35, indented twice: contents, dot, add, left parenthesis, square, right parenthesis, semicolon. Line 36, indented twice: contents, dot, add, left parenthesis, cube, right parenthesis, semicolon. Line 37, indented twice: contents, dot, add, left parenthesis, result Label, right parenthesis, semicolon. Line 38, indented twice: contents, dot, add, left parenthesis, result, right parenthesis, semicolon. Line 39: Blank. Line 40, indented twice: forward slash, forward slash, instantiate our event handler. Line 41, indented twice: Button Handler, b h, equals, new, Button Handler, left parenthesis, right parenthesis, semicolon. Line 42: Blank. Line 43, indented twice: Forward slash, forward slash, add event handler as listener for both buttons. Line 44, indented twice: square, dot, add Action Listener, left parenthesis, b h, right parenthesis, semicolon. Line 45, indented twice: cube, dot, add Action Listener, left parenthesis, b h, right parenthesis, semicolon. Line 46: Blank. Line 47, indented twice: set Size, left parenthesis, 175, comma, 150, right parenthesis, semicolon. Line 48, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 49, indented once: Right brace. Line 50: Blank. Line 51, indented once: forward slash, forward slash, private inner class event handler. Line 52, indented once: private, class, Button Handler, implements, Action Listener. Line 53, indented once: Left brace. Line 54, indented twice: forward slash, forward slash, implement action Performed method. Line 55, indented twice: public, void, action Performed, left parenthesis, Action Event, a e, right parenthesis. Line 56, indented twice: Left brace. Line 57, indented 3 times: try. Line 58, indented 3 times: Left brace. Line 59, indented 4 times: double, o p, equals, Double, dot, parse Double, left parenthesis, operand, dot, get Text, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 60: Blank. Line 61, indented 4 times: forward slash, forward slash, identify which button was pressed. Line 62, indented 4 times: if, left parenthesis, a e, dot, get Source, left parenthesis, right parenthesis, equals, equals, square, right parenthesis. Line 63, indented 5 times: result, dot, set Text, left parenthesis, Double, dot, value Of, left parenthesis, o p, asterisk, o p, right parenthesis, dot, to String, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 64, indented 4 times: else, if, left parenthesis, a e, dot, get Source, left parenthesis, right parenthesis, equals, equals, cube, right parenthesis. Line 65, indented 5 times: result, dot, set Text, left parenthesis, Double, dot, value Of, left parenthesis, o p, asterisk, o p, asterisk, o p, right parenthesis, dot, to String, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 66, indented 3 times: Right brace. Line 67, indented 3 times: catch, left parenthesis, Number Format Exception, e, right parenthesis. Line 68, indented 3 times: Left brace. Line 69, indented 4 times: operand Label, dot, set Text, left parenthesis, open quotes, Enter a number, close quotes, right parenthesis, semicolon. Line 70, indented 4 times: operand, dot, set Text, left parenthesis, open quotes, close quotes, right parenthesis, semicolon. Line 71, indented 4 times: result, dot, set Text, left parenthesis, open quotes, question mark, question mark, question mark, close quotes, right parenthesis, semicolon. Line 72, indented 3 times: Right brace. Line 73, indented twice: Right brace. Line 74, indented once: Right brace. Line 75: Blank. Line 76, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 77, indented once: Left brace. Line 78, indented twice: Simple Math, s m, equals, new, Simple Math, left parenthesis, right parenthesis, semicolon. Line 79, indented twice: s m, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 80, indented once: Right brace. Line 81: Right brace.

Back to Example

A graphical user interface in which color can be selected using radio buttons and checkboxes. The first section of the graphical user interface has the heading, Using Radio Buttons. Only one selection is possible at a time. Three columns of radio buttons are present for color selection with three radio buttons in each column. The columns are labeled red, green, and blue. The resulting color combination in each row of radio buttons is listed in the adjacent column that is labeled Resulting Color. The second section of the graphical user interface has the heading, Using Checkboxes. Zero to three selections are possible at a time. Three columns of checkboxes are present for color selection with eight checkboxes in each column. The columns are labeled red, green, and blue. The resulting color combination in each row of checkboxes is listed in the adjacent column that is labeled Resulting Color.

Back to Table

A table detailing useful constructors of the J Radio Button, Button Group, and J Check Box classes. Constructors of the J Radio Button, Button Group, and J Check Box Classes. Constructors. The table has 5 rows and 2 columns. It has the following column headings. Class. Constructor. The row details of the table are the following. Row 1: Class, J Radio Button; Constructor, J Radio Button, left parenthesis, String, button Label, right parenthesis, constructs a radio button labeled button Label. By default, the radio button is initially deselected. Row 2: Class, J Radio Button; Constructor, J Radio Button, left parenthesis, String, button Label, comma, boolean, selected, right parenthesis, constructs a radio button labeled button Label. If selected is true, the button is initially selected; if selected is false, the button is deselected. Row 3: Class, Button Group; Constructor, Button Group, left parenthesis, right parenthesis, constructs a button group. Adding buttons to this group makes the buttons mutually exclusive. Row 4: Class, J Check Box; Constructor, J Check Box, J Check Box, left parenthesis, String, check Box Label, right parenthesis, constructs a checkbox labeled check Box Label. By default, the checkbox is initially deselected. Row 5: Class, J Check Box; Constructor, J Check Box, left parenthesis, String, check Box Label, comma, boolean, selected, right parenthesis, constructs a checkbox labeled check Box Label. If selected is true, the checkbox is initially selected; if selected is false, the checkbox is initially deselected.

Back to Table

The Item Listener method A P I. public, void, item State Changed, left parenthesis, Item Event, event, right parenthesis. An event handler that implements the Item Listener interface writes code in this method to respond to the Item Event fired by any registered components.

Back to Table

Code for a program in which colors can be selected using J Radio Buttons. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Select a Color using J Radio Buttons. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x dot swing dot asterisk, semicolon. Line 6: import, java dot a w t dot asterisk, semicolon. Line 7: import, java dot a w t dot event dot asterisk, semicolon. Line 8: Blank. Line 9: public, class, Changing Colors, extends, J Frame. Line 10: Left brace. Line 11, indented once: private, Container, contents, semicolon. Line 12, indented once: private, J Radio Button, red, comma, green, comma, blue, semicolon. Line 13, indented once: private, Button Group, color Group, semicolon. Line 14, indented once: private, J Label, label, semicolon. Line 15, indented once: private, Color, selected Color, equals, Color dot RED, semicolon. Line 16: Blank. Line 17, indented once: public, Changing Colors, left parenthesis, right parenthesis. Line 18, indented once: Left brace. Line 19, indented twice: super, left parenthesis, open quotes, Selecting a color, close quotes, right parenthesis, semicolon. Line 20, indented twice: contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 21, indented twice: contents, dot, set Layout, left parenthesis, new, Flow Layout, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 22: Blank. Line 23, indented twice: red, equals, new, J Radio Button, left parenthesis, open quotes, red, close quotes, comma, true, right parenthesis, semicolon. Line 24, indented twice: green, equals, new, J Radio Button, left parenthesis, open quotes, green, close quotes, right parenthesis, semicolon. Line 25, indented twice: blue, equals, new, J Radio Button, left parenthesis, open quotes, blue, close quotes, right parenthesis, semicolon. Line 26: Blank. Line 27, indented twice: label, equals, new, J Label, left parenthesis, open quotes, Watch my background, close quotes, right parenthesis, semicolon. Line 28, indented twice: label, dot, set Foreground, left parenthesis, Color, dot, GRAY, right parenthesis, semicolon. Line 29, indented twice: label, dot, set Opaque, left parenthesis, true, right parenthesis, semicolon. Line 30, indented twice: label, dot, set Background, left parenthesis, selected Color, right parenthesis, semicolon. Line 31: Blank. Line 32, indented twice: contents, dot, add, left parenthesis, red, right parenthesis, semicolon. Line 33, indented twice: contents, dot, add, left parenthesis, green, right parenthesis, semicolon. Line 34, indented twice: contents, dot, add, left parenthesis, blue, right parenthesis, semicolon. Line 35, indented twice: contents, dot, add, left parenthesis, label, right parenthesis, semicolon. Line 36: Blank. Line 37, indented twice: Forward slash, forward slash, create button group. Line 38, indented twice: color Group, equals, new, Button Group, left parenthesis, right parenthesis, semicolon. Line 39, indented twice: color Group, dot, add, left parenthesis, red, right parenthesis, semicolon. Line 40, indented twice: color Group, dot, add, left parenthesis, green, right parenthesis, semicolon. Line 41, indented twice: color Group, dot, add, left parenthesis, blue, right parenthesis, semicolon. Line 42: Blank. Line 43, indented twice: forward slash, forward slash, create Radio Button Handler event handler. Line 44, indented twice: forward slash, forward slash, and register it on the radio buttons. Line 45, indented twice: Radio Button Handler, r b h, equals, new, Radio Button Handler, left parenthesis, right parenthesis, semicolon. Line 46, indented twice: red, dot, add Item Listener, left parenthesis, r b h, right parenthesis, semicolon. Line 47, indented twice: green, dot, add Item Listener, left parenthesis, r b h, right parenthesis, semicolon. Line 48, indented twice: blue, dot, add Item Listener, left parenthesis, r b h, right parenthesis, semicolon. Line 49: Blank. Line 50, indented twice: set Size, left parenthesis, 225, comma, 200, right parenthesis, semicolon. Line 51, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 52, indented once: Right brace. Line 53: Blank. Line 54, indented once: private, class, Radio Button Handler, implements, Item Listener. Line 55, indented once: Left brace. Line 56, indented twice: public, void, item State Changed, left parenthesis, Item Event, i e, right parenthesis. Line 57, indented twice: Left brace. Line 58, indented 3 times: if, left parenthesis, i e, dot, get Source, left parenthesis, right parenthesis, equals, equals, red, right parenthesis. Line 59, indented 4 times: selected Color, equals, Color, dot, RED, semicolon. Line 60, indented 3 times: else, if, left parenthesis, i e, dot, get

Source, left parenthesis, right parenthesis, equals, equals, green, right parenthesis. Line 61, indented 4 times: selected Color, equals, Color, dot, GREEN, semicolon. Line 62, indented 3 times: else, if, left parenthesis, i e, dot, get Source, left parenthesis, right parenthesis, equals, equals, blue, right parenthesis. Line 63, indented 4 times: selected Color, equals, Color, dot, BLUE, semicolon. Line 64: Blank. Line 65, indented 3 times: label, dot, set Background, left parenthesis, selected Color, right parenthesis, semicolon. Line 66, indented twice: Right brace. Line 67, indented once: Right brace. Line 68: Blank. Line 69, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 70, indented once: Left brace. Line 71, indented twice: Changing Colors, c c, equals, new, Changing Colors, left parenthesis, right parenthesis, semicolon. Line 72, indented twice: c c, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 73, indented once: Right brace. Line 74: Right brace.

Back to Example

Code for a program that uses J Check Boxes. Program code. In the code, the words in the variable names are merged. Line 1, forward slash, asterisk, Using J Check Boxes. Line 2 Anderson, comma, Franceschi. Line 3, asterisk, forward slash. Line 4: Blank. Line 5: import, java x dot swing dot asterisk, semicolon. Line 6: import, java dot a w t dot asterisk, semicolon. Line 7: import, java dot a w t dot event dot asterisk, semicolon. Line 8: Blank. Line 9: public, class, Mixing Colors, extends, J Frame. Line 10: Left brace. Line 11, indented once: private, Container, contents, semicolon. Line 12, indented once: private, J Check Box, red, comma, green, comma, blue, semicolon. Line 13, indented once: private, i n t, red Value, comma, green Value, comma, blue Value, semicolon. Line 14, indented once: private, J Label, label, semicolon. Line 15: Blank. Line 16, indented once: public, Mixing Colors, left parenthesis, right parenthesis. Line 17, indented once: Left brace. Line 18, indented once: super, left parenthesis, open quotes, Selecting a color, close quotes, right parenthesis, semicolon. Line 19, indented once: contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 20, indented once: contents, dot, set Layout, left parenthesis, new, Flow Layout, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 21: Blank. Line 22, indented once: red, equals, new, J Check Box, left parenthesis, open quotes, red, close quotes, semicolon, semicolon. Line 23, indented once: green, equals, new, J Check Box, left parenthesis, open quotes, green, close quotes, right parenthesis, semicolon. Line 24, indented once: blue, equals, new, J Check Box, left parenthesis, open quotes, blue, close quotes, right parenthesis, semicolon. Line 25: Blank. Line 26, indented once: label, equals, new, J Label, left parenthesis, open quotes, Watch my background, close quotes, right parenthesis, semicolon.

Line 27, indented once: label, dot, set Opaque, left parenthesis, true, right parenthesis, semicolon. Line 28, indented once: label, dot, set Foreground, left parenthesis, Color, dot, GRAY, right parenthesis, semicolon. Line 29, indented once: label, dot, set Background, left parenthesis, new, Color, left parenthesis, 0, comma, 0, comma, 0, right parenthesis, right parenthesis, semicolon. Line 30: Blank. Line 31, indented once: contents, dot, add, left parenthesis, red, right parenthesis, semicolon. Line 32, indented once: contents, dot, add, left parenthesis, green, right parenthesis, semicolon. Line 33, indented once: contents, dot, add, left parenthesis, blue, right parenthesis, semicolon. Line 34, indented once: contents, dot, add, left parenthesis, label, right parenthesis, semicolon. Line 35: Blank. Line 36, indented once: forward slash, forward slash, create Check Box Handler event handler. Line 37, indented once: forward slash, forward slash, and register it on the checkboxes. Line 38, indented once: Check Box Handler, c b h, equals, new, Check Box Handler, left parenthesis, right parenthesis, semicolon. Line 39, indented once: red, dot, add Item Listener, left parenthesis, c b h, right parenthesis, semicolon. Line 40, indented once: green, dot, add Item Listener, left parenthesis, c b h, right parenthesis, semicolon. Line 41, indented once: blue, dot, add Item Listener, left parenthesis, c b h, right parenthesis, semicolon. Line 42: Blank. Line 43, indented once: set Size, left parenthesis, 225, comma, 200, right parenthesis, semicolon. Line 44, indented once: set

Visible, left parenthesis, true, right parenthesis, semicolon. Line 45, indented once: Right brace. Line 46: Blank. Line 47, indented once: private, class, Check Box Handler, implements, Item Listener. Line 48, indented once: Left brace. Line 49, indented twice: public, void, item State Changed, left parenthesis, Item Event, i e, right parenthesis. Line 50, indented twice: Left brace. Line 51, indented 3 times: if, left parenthesis, i e, dot, get Source, left parenthesis, right parenthesis, equals, equals, red, right parenthesis. Line 52, indented 3 times: Left brace. Line 53, indented 4 times: if, left parenthesis, i e, dot, get State Change, left parenthesis, right parenthesis, equals, equals, Item Event, dot, SELECTED, right parenthesis. Line 54, indented 5 times: red Value, equals, 255, semicolon. Line 55, indented 4 times: else. Line 56, indented 5 times: red Value, equals, 0, semicolon. Line 57, indented 3 times: Right brace. Line 58, indented 3 times: else, if, left parenthesis, i e, dot, get Source, left parenthesis, right parenthesis, equals, equals, green, right parenthesis. Line 59, indented 3 times: Left brace. Line 60, indented 4 times: if, left parenthesis, i e, dot, get State Change, left parenthesis, right parenthesis, equals, equals, Item Event, dot, SELECTED, right parenthesis. Line 61, indented 5 times: green Value, equals, 255, semicolon. Line 62, indented 4 times: else. Line 63, indented 5 times: green Value, equals, 0, semicolon. Line 64, indented 3 times: Right brace. Line 65, indented 3 times: else, if, left parenthesis, i e, dot, get Source, left parenthesis, right parenthesis, equals, equals, blue, right parenthesis. Line 66, indented 3 times: Left brace. Line 67, indented 4 times: if, left parenthesis, i e, dot, get State Change, left parenthesis, right parenthesis, equals, equals, Item Event, dot, SELECTED, right parenthesis. Line 68, indented 5 times: blue Value, equals, 255, semicolon. Line 69, indented 4 times: else. Line 70, indented 5 times: blue Value, equals, 0, semicolon. Line 71, indented 3 times: Right brace. Line 72: Blank. Line 73, indented 3 times: label, dot, set Background, left parenthesis. Line 74, indented a few times: new, Color, left parenthesis, red Value, comma, green Value, comma, blue Value, right parenthesis, right parenthesis, semicolon. Line 75, indented twice: Right brace. Line 76, indented once: Right brace. Line 77: Blank. Line 78: indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 79, indented once. Left brace. Line 80, indented twice: Mixing Colors, m c, equals, new, Mixing Colors, left parenthesis, right parenthesis, semicolon. Line 81, indented twice: m c, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 82, indented once: Right brace. Line 83: Right brace.

Back to Example

A table detailing a useful method of the Item Event class. The return value of the method is i n t. The method name and argument list is the following. get State Change, left parenthesis, right parenthesis, returns the state of the checkbox. If the checkbox is selected, the value SELECTED is returned; if the checkbox is deselected, the value DESELECTED is returned, where SELECTED and DESELECTED are static i n t constants of the Item Event class.

Back to Table

The location of Student Code in the program J Button Practice dot java. Program code. In the code, the words in the variable names are merged. Line 1, indented once: dot, dot, dot. Line 2, indented once: open, equals, new, J Button, left parenthesis, open quotes, OPEN, close quotes, right parenthesis, semicolon. Line 3, indented once: contents, dot, add, left parenthesis, open, right parenthesis, semicolon. Line 4, indented once: close, equals, new, J Button, left parenthesis, open quotes, CLOSE, close quotes, right parenthesis, semicolon. Line 5, indented once: contents, dot, add, left parenthesis, close, right parenthesis, semicolon. Line 6, indented once: Forward slash, forward slash, a few asterisks, 1. Student code starts here. Line 7, indented once: forward slash, forward slash, declare and instantiate the button handler. Line 8, indented once: forward slash, forward slash, and register it on the buttons. Line 9, indented once: forward slash, forward slash, end of task 1. Line 10, indented once: set Size, left parenthesis, 500, comma, 375, right parenthesis, semicolon. Line 11, indented once: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 12: Right brace. Line 13: forward slash, forward slash, a few asterisks, 2. Student code restarts here. Line 14: forward slash, forward slash, Code a private class to implement the correct Listener. Line 15: forward slash, forward slash, and its required method. Line 16: forward slash, forward slash, To open the switch, call the open method with the statement. Line 17: forward slash, forward slash, open, left parenthesis, right parenthesis, semicolon. Line 18: forward slash, forward slash, To close the switch, call the close method with the statement. Line 19: forward slash, forward slash, close, left parenthesis, right parenthesis, semicolon. Line 20: forward slash, forward slash, The last statement of the method should be. Line 21: forward slash, forward slash, animate, left parenthesis, right parenthesis, semicolon. Line 22: forward slash, forward slash, end of task 2.

Back to Example

A table detailing useful constructors and methods of the J Combo Box class. Constructor. Class: J Combo Box. Constructor. J Combo Box, left parenthesis, Object, left square brace, right square brace, array Name, right parenthesis, constructs a new J Combo Box component initially filled with the objects in array Name. Often, the objects are Strings. Methods. The table has 3 rows and 2 columns. The column headings in the table are the following. Return value. Method name and argument list. The row details in the table are the following. Row 1: Return value, void; Method name and argument list, set Selected Index, left parenthesis, i n t, index, right parenthesis, sets the item at index as selected. The index of the first item in the list is 0. Row 2: Return value, i n t; Method name and argument list, get Selected Index, left parenthesis, right parenthesis, returns the index of the selected item. The index of the first item in the list is 0. Row 3: Return value, void; Method name and argument list, set Maximum Row Count, left parenthesis, i n t, size, right parenthesis, sets the number of rows that will be visible at one time to size.

Back to Table

Code for a program that uses J Combo Box to display food samplings. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Using J Combo Box to show a sampling of international foods. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x, dot, swing, dot, asterisk, semicolon. Line 6: import, java, dot, a w t, dot, asterisk, semicolon. Line 7: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 8: Blank. Line 9: public, class, Food Samplings, extends, J Frame. Line 10: Left brace. Line 11, indented once: private, Container, contents, semicolon. Line 12, indented once: private, J Combo Box, left angle bracket, String, right angle bracket, countries, semicolon. Line 13, indented once: private, J Label, food Image, semicolon. Line 14: Blank. Line 15, indented once: private, String, left square brace, right square brace, country List, equals, Line 16, indented a few times: Left brace, open quotes, France, close quotes, comma, open quotes, Greece, close quotes, comma, open quotes, Italy, close quotes, comma, open quotes, Japan, close quotes, comma, open quotes, U S A, close quotes, right brace, semicolon. Line 17, indented once: private, Image Icon, left square brace, right square brace, foods, equals. Line 18, indented a few times: Left brace, new Image Icon, open quotes, cheese, dot, j p g, close quotes, right parenthesis, comma. Line 19, indented a few times: new, Image Icon, left parenthesis, open quotes, feta Salad, dot, j p g, close quotes, right parenthesis, comma. Line 20, indented a few times: new, Image Icon, left parenthesis, open quotes, pizza, dot, j p g, close quotes, right parenthesis, comm. Line 21, indented a few times: new, Image Icon, left parenthesis, open quotes, sushi, dot, j p g, close quotes, right parenthesis, comma. Line 22, indented once: new, Image Icon, left parenthesis, open quotes, hamburger, dot, j p g, close quotes, right parenthesis, right brace, semicolon. Line 23: Blank. Line 24, indented once: public, Food Samplings, left parenthesis, right parenthesis. Line 25, indented once: Left brace. Line 26, indented twice: super, left parenthesis, open quotes, Food samplings of various countries, close quotes, right parenthesis, semicolon. Line 27, indented twice: contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 28, indented twice: contents, dot, set Layout, left parenthesis, new, Flow Layout, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 29: Blank. Line 30, indented twice: forward slash, forward slash, instantiate the components. Line 31, indented twice: countries, equals, new, J Combo Box, left angle bracket, String, right angle bracket, left parenthesis, country List, right parenthesis, semicolon. Line 32, indented twice: countries, dot, set Maximum Row Count, left parenthesis, 4, right parenthesis, semicolon. Line 33, indented twice: countries, dot, set Selected Index, left parenthesis, 2, right parenthesis, semicolon. Line 34, indented twice: food Image, equals, new, J Label, left parenthesis, foods, left square brace, 2, right square brace, right parenthesis, semicolon. Line 35: Blank. Line 36, indented twice: forward slash, forward slash, add components to the content pane. Line 37, indented twice: contents, dot, add, left parenthesis, countries, right parenthesis, semicolon. Line 38, indented twice: contents, dot, add, left parenthesis, food Image, right parenthesis, semicolon. Line 39: Blank. Line 40, indented twice: forward slash, forward slash, set up event handler. Line 41, indented twice: Item Listener Handler, i l h, equals, new, Item Listener Handler, left parenthesis, right parenthesis, semicolon. Line 42, indented twice: countries, dot, add Item Listener, left parenthesis, i l h, right parenthesis, semicolon. Line 43: Blank. Line 44, indented twice: set Size, left parenthesis, 350, comma, 150, right parenthesis, semicolon. Line 45, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 46, indented once: right brace. Line 47: Blank. Line 48, indented once: private, class, Item

Listener Handler, implements, Item Listener. Line 49, indented once: left brace. Line 50, indented twice: public, void, item State Changed, left parenthesis, Item Event, i e, right parenthesis, Line 51, indented twice: Left brace. Line 52, indented 3 times: i n t, index, equals, countries, dot, get Selected Index, left parenthesis, right parenthesis, semicolon. Line 53, indented 3 times: food Image, dot, set Icon, left parenthesis, foods, left square brace, index, right square brace, right parenthesis, semicolon. Line 54, indented twice: Right brace. Line 55, indented once: Right brace. Line 56: Blank. Line 57, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 58, indented once: left brace. Line 59, indented twice: Food Samplings, f s, equals, new, Food Samplings, left parenthesis, right parenthesis, semicolon. Line 60, indented twice: f s, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 61, indented once: Right brace. Line 62: Right brace.

Back to Example

A table detailing useful methods of the Mouse Event class. The table has 2 rows and 2 columns. It has the following column headings. Return value. Method name and argument list. The row details in the table are the following. Row 1: Return value, i n t; Method name and argument list, get X, left parenthesis, right parenthesis, returns the x value of the (x, comma, y) coordinate of the mouse activity. Row 2: Return value, i n t; Method name and argument list, get Y, left parenthesis, right parenthesis, returns the y value of the (x, comma, y) coordinate of the mouse activity.

Back to Table

A table detailing methods of the Mouse Listener interface. Mouse Listener Interface Method A P Is. The row details in the table are the following. Row 1: public, void, mouse Pressed, left parenthesis, Mouse Event, e, right parenthesis, called when the mouse button is pressed. Row 2: public, void, mouse Released, left parenthesis, Mouse Event, e, right parenthesis, called when the mouse button is released after being pressed. Row 3: public, void, mouse Clicked, left parenthesis, Mouse Event, e, right parenthesis, called when the mouse button is pressed and released. Row 4: public, void, mouse Entered, left parenthesis, Mouse Event, e, right parenthesis, called when the mouse enters the registered component. Row 5: public, void, mouse Exited, left parenthesis, Mouse Event, e, right parenthesis, called when the mouse exits the registered component.

Back to Table

Code for the Sub Hunt class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Sub Hunt class. Line 2 *

Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java, dot, a w t, dot, Graphics, semicolon. Line 6: import, java, dot, a w t, dot, Color, semicolon. Line 7: import, java, dot, util, dot, Random, semicolon. Line 8: Blank. Line 9: public, class, Sub Hunt. Line 10: Left brace. Line 11, indented once: public, static, i n t, DEFAULT underscore GAME underscore SIZE, equals, 300, semicolon. Line 12, indented once: public, static, i n t, SIDE, equals, 36, semicolon, forward slash, forward slash, size of submarine. Line 13: Blank. Line 14, indented once: private, i n t, game Size, semicolon. Line 15, indented once: private, i n t, x C t r, semicolon, forward slash, forward slash, x coordinate of center of submarine. Line 16, indented once: private, i n t, y C t r, semicolon, forward slash, forward slash, y coordinate of center of submarine. Line 17, indented once: private, String, status, equals, open quotes, close quotes, semicolon. Line 18, indented once: private, boolean, hit, semicolon. Line 19: Blank. Line 20, indented once: Forward slash, asterisk, asterisk, Constructor. Line 21, indented once: asterisk, at, param, new Game Size, game size. Line 22, indented once: asterisk, forward slash. Line 23, indented once: public, Sub Hunt, left parenthesis, i n t, new Game Size, right parenthesis. Line 24, indented once: Left brace. Line 25, indented twice: if, left parenthesis, new Game Size, greater than, SIDE, right parenthesis. Line 26, indented 3 times: game Size, equals, new Game Size, semicolon. Line 27, indented twice: else. Line 28, indented 3 times:

game Size, equals, DEFAULT underscore GAME underscore SIZE, semicolon. Line 29, indented twice: Forward slash, forward slash, generate submarine center. Line 30, indented twice: Random, random, equals, new, Random, left parenthesis, right parenthesis, semicolon. Line 31, indented twice: x C t r, equals, SIDE, forward slash, 2, plus, random, dot, next I n t, left parenthesis, game Size, minus, SIDE, right parenthesis, semicolon. Line 32, indented twice: y C t r, equals, SIDE, forward slash, 2, plus, random, dot, next I n t, left parenthesis, game Size, minus, SIDE, right parenthesis, semicolon. Line 33, indented twice: hit, equals, false, semicolon. Line 34, indented once: Right brace. Line 35: Blank. Line 36, indented once: forward slash, asterisk, asterisk, get Status. Line 37, indented once: asterisk, at, return, status. Line 38, indented once: asterisk, forward slash. Line 39, indented once: public, String, get Status, left parenthesis, right parenthesis. Line 40, indented once: Left brace. Line 41, indented twice: return, status, semicolon. Line 42, indented once: Right brace. Line 43: Blank. Line 44, indented once: forward slash, asterisk, asterisk, get Game Size. Line 45, indented once: asterisk, at return, game Size. Line 46, indented once: asterisk, forward slash. Line 47, indented once: public, i n t, get Game Size, left parenthesis, right parenthesis. Line 48, indented once: Left brace. Line 49, indented twice: return, game Size, semicolon. Line 50, indented once: Right brace. Line 51: Blank. Line 52, indented once: Forward slash, asterisk, asterisk, is Hit. Line 53, indented once: asterisk, at, return, hit. Line 54, indented once: asterisk, forward slash. Line 55, indented once: public, boolean, is Hit, left parenthesis, right parenthesis. Line 56, indented once: Left brace. Line 57, indented twice: return, hit, semicolon. Line 58, indented once: Right brace. Line 59: Blank. Line 60, indented once: Forward slash, asterisk, asterisk, play. Line 61, indented once: Asterisk, at, param, x, the x coordinate of the play. Line 62, indented once: Asterisk, at, param, y, the y coordinate of the play. Line 63, indented once: Asterisk, forward slash. Line 64, indented once: public, void, play, left parenthesis, i n t, x, comma, i n t, y, right parenthesis. Line 65, indented once: Left brace. Line 66, indented twice: Forward slash, forward slash, is click within the submarine? Line 67, indented twice: if, left parenthesis, Math, dot, a b s, left parenthesis, x, minus, x C t r, right parenthesis, less than, SIDE, forward slash, 2. Line 68, indented a few times: ampersand, ampersand, Math, dot, a b s, left parenthesis, y, minus, y C t r, right parenthesis, less than, SIDE, forward slash, 2, right parenthesis. Line 69, indented twice: Left brace. Line 70, indented 3 times: status, equals, open quotes, Sunk, exclamation mark, close quotes, semicolon. Line 71, indented 3 times: hit, equals, true, semicolon. Line 72, indented twice: Right brace. Line 73, indented twice: Forward slash, forward slash, is click close? Line 74, indented twice: else, if, left parenthesis, Math, dot, a b s, left parenthesis, x, minus, x C t r, right parenthesis, less than, 2, asterisk, SIDE. Line 75, indented a few times: ampersand, ampersand, Math, dot, a b s, left parenthesis, y, minus, y C t r, right parenthesis, less than, 2, asterisk, SIDE, right parenthesis. Line 76, indented 3 times: status, equals, open quotes, Close, dot, dot, dot, close quotes, semicolon. Line 77, indented twice: Forward slash, forward slash, click is too far from submarine. Line 78, indented twice: else. Line 79, indented 3 times: status, equals, open quotes, In the water, close quotes, semicolon. Line 80, indented once: Right brace. Line 81: Blank. Line 82, indented once: forward slash, asterisk, asterisk, draw. Line 83, indented once: Asterisk, at, param, g a, Graphics object. Line 84, indented once: Asterisk, at, param, x, the x coordinate of the play. Line 85, indented once: Asterisk, at, param, y, the y coordinate of the play. Line 86, indented once: Asterisk, forward slash. Line 87, indented once: public, void, draw, left parenthesis, Graphics, g, comma, i n t, x, comma, i n t, y, right parenthesis. Line 88, indented once: Left brace. Line 89, indented twice: if, left parenthesis, status, dot, equals, left parenthesis, open quotes, Sunk, exclamation mark, close quotes, right parenthesis, right parenthesis. Line 90, indented twice: Left brace. Line 91, indented 3 times: Forward slash, forward slash, draw sunken submarine. Line 92, indented 3 times: g, dot, set Color, left parenthesis, Color, dot, BLACK, right parenthesis, semicolon. Line 93, indented 3 times: g, dot, fill Round Rect, left parenthesis, x C t r, minus, SIDE, forward slash, 2, comma, y C t r, minus, SIDE, forward slash, 2, comma. Line 94, indented a few times: SIDE, forward slash, 2, comma, SIDE, comma, SIDE, forward slash, 2, comma, SIDE, forward slash, 2, right parenthesis, semicolon. Line 95: Blank. Line 96, indented a 3 times: g, dot, fill Round Rect, left parenthesis, x C t r, minus, SIDE, forward slash, 4, comma, y C t r, minus, SIDE, forward slash, 3, comma. Line 97, indented a few times: SIDE, forward slash, 2, SIDE, forward slash, 2, comma, SIDE, forward slash, 4, comma, SIDE, forward slash, 4, right parenthesis, semicolon. Line 98: Blank. Line 99, indented 3 times: g, dot, draw Line, left parenthesis, x C t r, plus, SIDE, forward slash, 4, comma, y C t r, minus, SIDE, forward slash, 9, comma. Line 100, indented a few times: x C t r, plus, SIDE, forward slash, 2, comma, y C t r, minus, SIDE, forward slash, 9, right parenthesis, semicolon. Line 101: Blank. Line 102, indented 3 times: forward slash, forward slash, draw red depth charge. Line 103, indented 3 times: g, dot, set Color, left parenthesis, Color, dot, RED, right parenthesis, semicolon. Line 104, indented a few times: g, dot, fill Oval, left parenthesis, x, minus, SIDE, forward slash, 4, comma, y, minus, SIDE, forward slash, 4, comma, SIDE, forward slash, 2, comma, SIDE, forward slash, 2, right parenthesis, semicolon. Line 105, indented twice: Right brace. Line 106, indented twice: else, if, left parenthesis, status, dot, equals, left parenthesis, open quotes, In the water, close quotes, right parenthesis, right parenthesis, forward slash, forward slash, draw blue circle. Line 107, indented twice: Left brace. Line 108, indented 3 times: g, dot, set Color, left parenthesis, Color, dot, BLUE, right parenthesis, semicolon. Line 109, indented 3 times: g, dot, fill Oval, left parenthesis, x, minus, SIDE, forward slash, 2, comma, y, minus, SIDE, forward slash, 2, comma, SIDE, comma, SIDE, right parenthesis, semicolon. Line 110, indented twice: Right brace. Line 111, indented twice: forward slash, forward slash, else, Close, dot, dot, dot, comma, do not draw. Line 112, indented once: Right brace. Line 113: Right brace.

Back to Example

Code for the Sub Hunt View Controller class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Sub Hunt View Controller class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x, dot, swing, dot, asterisk, semicolon. Line 6: import, java, dot, a w t, dot, asterisk, semicolon. Line 7: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 8: Blank. Line 9: public, class, Sub Hunt View Controller, extends, J Frame. Line 10: Left brace. Line 11, indented once: private, i n t, x, semicolon, forward slash, current x mouse position. Line 12, indented once: private, i n t, y, semicolon, forward slash, forward slash, current y mouse position. Line 13, indented once: private, Sub Hunt, sub, semicolon, forward slash, forward slash, submarine. Line 14, indented once: private, boolean, game Started, equals, false, semicolon. Line 15, indented once: private, Insets, insets, semicolon. Line 16, indented once: private, Mouse Handler, m h, semicolon, forward slash, forward slash, mouse event handler. Line

17: Blank. Line 18, indented once: public, Sub Hunt View Controller, left parenthesis, Sub Hunt, new Sub, right parenthesis. Line 19, indented once: Left brace. Line 20, indented twice: super, left parenthesis, open quotes, Play, exclamation mark, close quotes, right parenthesis, semicolon. Line 21: Blank. Line 22, indented twice: sub, equals, new Sub, semicolon. Line 23, indented twice: forward slash, forward slash, instantiate event handler and register listener on window. Line 24, indented twice: m h, equals, new, Mouse Handler, left parenthesis, right parenthesis, semicolon. Line 25, indented twice: add Mouse Listener, left parenthesis, m h, right parenthesis, semicolon. Line 26: Blank. Line 27, indented twice: forward slash, forward slash, size window to include insets. Line 28, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 29, indented twice: insets, equals, get Insets, left parenthesis, right parenthesis, semicolon. Line 30, indented twice: set Size, left parenthesis, sub, dot, get Game Size, left parenthesis, right parenthesis, plus, insets, dot, left, plus, insets, dot, right, comma. Line 31, indented a few times: sub, dot, get Game Size, left parenthesis, right parenthesis, plus, insets, dot, top, plus, insets, dot, bottom, right parenthesis, semicolon. Line 32, indented once: Right brace. Line 33: Blank. Line 34, indented once: private, class, Mouse Handler, extends, Mouse Adapter. Line 35, indented once: Left brace. Line 36, indented twice: public, void, mouse Clicked, left parenthesis, Mouse Event, m e, right parenthesis. Line 37, indented twice: Left brace. Line 38, indented 3 times: x, equals, m e, dot, get X, left parenthesis, right parenthesis, minus, insets, dot, left, semicolon. Line 39, indented 3 times: y, equals, m e, dot, get Y, left parenthesis, right parenthesis, minus, insets, dot, top, semicolon. Line 40, indented 3 times: sub, dot, play, left parenthesis, x, comma, y, right parenthesis, semicolon. Line 41, indented 3 times: set Title, left parenthesis, sub, dot, get Status, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 42, indented 3 times: repaint, left parenthesis, right parenthesis, semicolon. Line 43, indented twice: Right brace. Line 44, indented once: Right brace. Line 45: Blank. Line 46, indented once: public, void, paint, left parenthesis, Graphics, g, right parenthesis. Line 47, indented once: Left brace. Line 48, indented twice: if, left parenthesis, exclamation mark, game Started, right parenthesis. Line 49, indented twice: Left brace. Line 50, indented 3 times: super, dot, paint, left parenthesis, g, right parenthesis, semicolon. Line 51, indented 3 times: game Started, equals, true, semicolon. Line 52, indented twice: Right brace. Line 53, indented twice: g, dot, translate, left parenthesis, insets, dot, left, comma, insets, dot, top, right parenthesis, semicolon. Line 54, indented twice: sub, dot, draw, left parenthesis, g, comma, x, comma, y, right parenthesis, semicolon. Line 55, indented twice: if, left parenthesis, sub, dot, is Hit, left parenthesis, right parenthesis, right parenthesis. Line 56, indented 3 times: remove Mouse Listener, left parenthesis, m h, right parenthesis, semicolon. Line 57, indented twice: Right brace. Line 58: Right brace.

Back to Example

A table detailing the get Insets method inherited from the Container class. Method: Insets, get Insets, left parenthesis, right parenthesis. Description: Returns an object representing the insets of this Container.

Back to Table

A table detailing the instance variables of the Insets class. The table has 4 rows and 2 columns. It has the following column headings. Instance Variables. Description. The row details in the table are the following. Row 1: Instance Variable, public, i n t top; Description, The size of the top border. Row 2: Instance Variable, public, i n t, left; Description, The size of the left border. Row 3: Instance Variable, public, i n t, bottom; Description, The size of the bottom border. Row 4: Instance Variable, public, i n t, right; Description, The size of the right border.

Back to Table

A table detailing the repaint method A P I in the component class. The return value is void. The method name and argument list is repaint, left parenthesis, right parenthesis. This method automatically forces a call to the paint method.

Back to Table

A table detailing the translate method of the Graphics class. The method is void, translate, left parenthesis, i n t, x, comma, i n t, y, right parenthesis. This method translates the origin of the graphics context to the point x, comma, y in the current coordinate system.

Back to Table

Code for the Play Sub Hunt class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Play Sub Hunt class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x, dot, swing, dot, J Frame, semicolon. Line 6: Blank. Line 7: public, class, Play Sub Hunt. Line 8: Left brace. Line 9, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 10, indented once: Left brace. Line 11, indented twice: Sub Hunt, sub, equals, new, Sub Hunt, left parenthesis, Sub Hunt, dot, DEFAULT underscore GAME underscore SIZE, right parenthesis, semicolon. Line 12, indented once: Sub Hunt View Controller, sub G u i, equals, new, Sub Hunt View Controller, left parenthesis, sub, right parenthesis, semicolon. Line 13, indented once: sub G u i, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 14, indented once: Right brace. Line 15: Right brace.

Back to Example

A table detailing the Mouse Motion Listener Interface method A P Is. Row 1: public, void, mouse Moved, left parenthesis, Moue Event, e, right parenthesis, called when the mouse is moved onto a component. Row 2: public, void, mouse Dragged, left parenthesis, Mouse Event, e, right parenthesis, called when the mouse button is pressed on a component and the mouse is dragged.

Back to Table

Code for the Treasure Hunt class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Treasure Hunt class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java, dot, a w t, dot, asterisk, semicolon. Line 6: import, java, dot, util, dot, Random, semicolon. Line 7: Blank. Line 8: public, class, Treasure Hunt. Line 9: Left brace. Line 10, indented once: public, static, i n t, DEFAULT underscore GAME underscore SIZE, equals, 300, semicolon, forward slash, forward slash, side of window. Line 11, indented once: public, static, i n t, SIDE, equals, 40, semicolon, forward slash, forward slash, side of treasure. Line 12: Blank. Line 13, indented once: private, i n t, game Size, semicolon. Line 14, indented once: private, i n t, x C t r, semicolon, forward slash, forward slash, x coordinate of center of treasure. Line 15, indented once: private, i n t, y C t r, semicolon, forward slash, forward slash, y coordinate of center of treasure. Line 16, indented once: private, String, status, equals, open quotes, close quotes, semicolon, forward slash, forward slash, message. Line 17, indented once: private, boolean, game Over, equals, false, semicolon. Line 18: Blank. Line 19, indented once: Forward slash, asterisk, asterisk, Constructor. Line 20, indented once: Asterisk, forward slash. Line 21, indented once: public, Treasure Hunt, left parenthesis, i n t, new Game Size, right parenthesis. Line 22, indented once: Left brace. Line 23, indented twice: if, left parenthesis, new Game Size, greater than, SIDE, right parenthesis. Line 24, indented 3 times: game Size, equals, new Game Size, semicolon. Line 25, indented twice: else. Line 26, indented 3 times: game Size, equals, DEFAULT underscore GAME underscore SIZE, semicolon. Line 27, indented twice: forward slash, forward slash, generate treasure center. Line 28, indented twice: Random, random, equals, new, Random, left parenthesis, right parenthesis, semicolon. Line 29, indented twice: x C t r, equals, SIDE, forward slash, 2, plus, random, dot, next I n t, left parenthesis, game Size, minus, SIDE, right parenthesis, semicolon. Line 30, indented twice: y C t r, equals, SIDE, forward slash, 2, plus, random, dot, next I n t, left parenthesis, game Size, minus, SIDE, right parenthesis, semicolon. Line 31, indented once: Right brace. Line 32: Blank. Line 33, indented once: forward slash, asterisk, asterisk, get Game Size. Line 34, indented once: asterisk, at, return, game Size. Line 35, indented once: asterisk, forward slash. Line 36, indented once:

public, i n t, get Game Size, left parenthesis, right parenthesis. Line 37, indented once: Left brace. Line 38, indented twice: return, game Size, semicolon. Line 39, indented once: Right brace. Line 40: Blank. Line 41, indented once: forward slash, asterisk, asterisk, is Game Over. Line 42, indented once: asterisk, @, return, game Over. Line 43, indented once: Asterisk, forward slash. Line 44, indented once: public, boolean, is Game Over, left parenthesis, right parenthesis. Line 45, indented once: Left brace. Line 46, indented twice: return, game Over, semicolon. Line 47, indented once: Right brace. Line 48: Blank. Line 49, indented once: forward slash, asterisk, asterisk, play. Line 50, indented once: asterisk, at, param, x, the x coordinate of the play. Line 51, indented once: asterisk, at, param, y, the y coordinate of the play. Line 52, indented once: asterisk, forward slash. Line 53, indented once: public, void, play, left parenthesis, i n t, x, comma, i n t, y, right parenthesis. Line 54, indented once: Left brace. Line 55, indented twice: forward slash, forward slash, is mouse within treasure? Line 56, indented twice: if, left parenthesis, Math, dot, a b s, left parenthesis, x, minus, x C t r, right parenthesis, less than, SIDE, forward slash, 2. Line 57, indented a few times: ampersand, ampersand, Math, dot, a b s, left parenthesis, y, minus, y C t r, right parenthesis, less than, SIDE, forward slash, 2, right parenthesis. Line 58, indented twice: Left brace. Line 59, indented 3 times: game Over, equals, true, semicolon. Line 60, indented 3 times: status, equals, open quotes, Found, close quotes, semicolon. Line 61, indented twice:

Right brace. Line 62, indented twice: forward slash, forward slash, is mouse within half-length of the treasure? Line 63, indented twice: else, if, left parenthesis, Math, dot, a b s, left parenthesis, x, minus, x C t r, right parenthesis, less than, left parenthesis, 1.5, asterisk, SIDE, right parenthesis. Line 64, indented a few tiems: ampersand, ampersand, Math, dot, a b s, left parenthesis, y, minus, y C t r, right parenthesis, less than, left parenthesis, 1.5, asterisk, SIDE, right parenthesis, right parenthesis. Line 65, indented 3 times: status, equals, open quotes, Hot, close quotes, semicolon. Line 66, indented twice: forward slash, forward slash, is mouse within 1 length of the treasure? Line 67, indented twice: else, if, left parenthesis, Math, dot, a b s, left parenthesis, x, minus, x C t r, right parenthesis, less than, left parenthesis, 2, asterisk, SIDE, right parenthesis. Line 68, indented a few times: ampersand, ampersand, Math, dot, a b s, left parenthesis, y, minus, y C t r, right parenthesis, less than, left parenthesis, 2, asterisk, SIDE, right parenthesis, right parenthesis. Line 69, indented 3 times: status, equals, open quotes, Warm, close quotes, semicolon. Line 70, indented twice: forward slash, forward slash, is mouse within 2 lengths of the treasure? Line 71, indented once: else, if, left parenthesis, Math, dot, a b s, left parenthesis, x, minus, x C t r, right parenthesis, less than, left parenthesis, 3, asterisk, SIDE, right parenthesis. Line 72, indented a few times: ampersand, ampersand, Math, dot, a b s, left parenthesis, y, minus, y C t r, right parenthesis, less than, left parenthesis, 3, asterisk, SIDE, right parenthesis, right parenthesis. Line 73, indented 3 times: status, equals, open quotes, Lukewarm, close quotes, semicolon. Line 74, indented twice: else, forward slash, forward slash, mouse is not near treasure. Line 75, indented 3 times: status, equals, open quotes, Cold, close quotes, semicolon. Line 76, indented once: Right brace. Line 77: Blank. Line 78, indented once: forward slash, asterisk, asterisk, draw. Line 79, indented once: asterisk, at, param, g, a Graphics object. Line 80, indented once: asterisk, at, param, x, the x coordinate of the play. Line 81, indented once: Asterisk, at, param, y, the y coordinate of the play. Line 82, indented once: Asterisk, forward slash. Line 83, indented once: public, void, draw, left parenthesis, Graphics, g, comma, i n t, x, comma, i n t, y, right parenthesis. Line 84, indented once: Left brace. Line 85, indented twice: g, dot, set Color, left parenthesis, Color, dot, BLUE, right parenthesis, semicolon. Line 86: Blank. Line 87, indented twice: if, left parenthesis, status, dot, equals, left parenthesis, open quotes, Found, close quotes, right parenthesis, right parenthesis, forward slash, forward slash, if found, comma, draw treasure. Line 88, indented twice: Left brace. Line 89, indented 3 times: g, dot, set Color, left parenthesis, Color, dot, RED, right parenthesis, semicolon. Line 90, indented 3 times: g, dot, fill Rect, left parenthesis, x C t r, minus, SIDE, forward slash, 2, comma, y C t r, minus, SIDE, forward slash, 2, comma, SIDE, comma, SIDE, right parenthesis, semicolon. Line 91, indented 3 times: g, dot, set Color, left parenthesis, Color, dot, GREEN, right parenthesis, semicolon. Line 92, indented 3 times: g, dot, draw String, left parenthesis, open quotes, dollar, dollar, dollar, close quotes, comma, x C t r, minus, SIDE, forward slash, 4, comma, y C t r, right parenthesis, semicolon. Line 93, indented twice: Right brace. Line 94, indented twice: else. Line 95, indented 3 times: g, dot, draw String, left parenthesis, status, comma, x, comma, y, right parenthesis, semicolon, forward slash, forward slash, display current status. Line 96, indented once: Right brace. Line 97: Right brace.

Back to Example

Code for the Treasure Hunt View Controller class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Treasure Hunt View Controller class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x, dot, swing, dot, J Frame, semicolon. Line 6: import, java, dot, a w t, dot, asterisk, semicolon. Line 7: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 8: Blank. Line 9: public, class, Treasure Hunt View Controller, extends, J Frame. Line 10, indented a few times: implements, Mouse Motion Listener. Line 11: Left brace. Line 12, indented once: private, i n t, x, semicolon. Line 13, indented once: private, i n t, y, semicolon. Line 14, indented once: private, Treasure Hunt, hunt, semicolon. Line 15, indented once: private, Insets, insets, semicolon. Line 16: Blank. Line 17, indented once: public, Treasure Hunt View

Controller, left parenthesis, Treasure Hunt, new Hunt, right parenthesis. Line 18, indented once: Left brace. Line 19, indented twice: super, left parenthesis, open quotes, Play, exclamation mark, close quotes, right parenthesis, semicolon. Line 20: Blank. Line 21, indented twice: hunt, equals, new Hunt, semicolon. Line 22, indented twice: forward slash, forward slash, application registers on itself. Line 23, indented twice: forward slash, forward slash, since it is a Mouse Motion Listener itself. Line 24, indented twice: add Mouse Motion Listener, left parenthesis, this, right parenthesis, semicolon. Line 25, indented twice: set Size, left parenthesis, hunt, dot, get Game Size, left parenthesis, right parenthesis, comma, hunt, dot, get Game Size, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 26, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 27: Blank. Line 28, indented twice: forward slash, forward slash, resize window to include insets. Line 29, indented twice: insets, equals, get Insets, left parenthesis, right parenthesis, semicolon. Line 30, indented twice: set Size, left parenthesis, get Width, left parenthesis, right parenthesis, plus, insets, dot, left, plus, insets, dot, right, comma. Line 31, indented a few times: get Height, left parenthesis, right parenthesis, plus, insets, dot, top, plus, insets, dot, bottom, right parenthesis, semicolon. Line 32, indented once: Right brace. Line 33: Blank. Line 34, indented once: public, void, mouse Moved, left parenthesis, Mouse Event, m e, right parenthesis. Line 35, indented once: Left brace. Line

36, indented twice: forward slash, forward slash, get location of mouse. Line 37, indented twice: x, equals, m e, dot, get X, left parenthesis, right parenthesis, minus, insets, dot, left, semicolon. Line 38, indented twice: y, equals, m e, dot, get Y, left parenthesis, right parenthesis, minus, insets, dot, top, semicolon. Line 39, indented twice: hunt, dot, play, left parenthesis, x, comma, y, right parenthesis, semicolon. Line 40, indented twice: repaint, left parenthesis, right parenthesis, semicolon. Line 41, indented twice: if, left parenthesis, hunt, dot, is Game Over, left parenthesis, right parenthesis, right parenthesis. Line 42, indented 3 times: remove Mouse Motion Listener, left parenthesis, this, right parenthesis, semicolon. Line 43, indented once: Right brace. Line 44: Blank. Line 45, indented once: public, void, mouse Dragged, left parenthesis, Mouse Event, m e, right parenthesis. Line 46, indented once: Left brace, right brace, forward slash, forward slash, we do not want to process mouse drag events. Line 47: Blank. Line 48, indented once: public, void, paint, left parenthesis, Graphics, g, right parenthesis. Line 49, indented once: Left brace. Line 50, indented twice: super, dot, paint, left parenthesis, g, right parenthesis, semicolon. Line 51, indented twice: g, dot, translate, left parenthesis, insets, dot, left, comma, insets, dot, top, right parenthesis, semicolon. Line 52, indented twice: hunt, dot, draw, left parenthesis, g, comma, x, comma, y, right parenthesis, semicolon. Line 53, indented once: Right brace. Line 54: Right brace.

Back to Example

Code for the Play Treasure Hunt class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Play Treasure Hunt class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x, dot, swing, dot, J Frame, semicolon. Line 6: Blank. Line 7: public, class, Play Treasure Hunt. Line 8: Left brace. Line 9, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 10, indented once: Left brace. Line 11, indented twice: Treasure Hunt, t h, equals, new, Treasure Hunt, left parenthesis. Line 12, indented a few times: Treasure Hunt, dot, DEFAULT underscore GAME underscore SIZE, right parenthesis, semicolon. Line 13, indented twice: Treasure Hunt View Controller, hunt G u i. Line 14, indented 3 times: equals, new, Treasure Hunt View Controller, left parenthesis, t h, right parenthesis, semicolon. Line 15, indented twice: hunt G u i, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 16, indented once: Right brace. Line 17: Right brace.

Back to Example

A table detailing Grid Layout constructors. The table has two rows and two columns. It has the following column headings. Class. Constructor. The row details are the following. Row 1: Class, Grid Layout; Constructor, Grid Layout, left parenthesis, i n t, number Of Rows, comma, i n t, number Of Columns, right parenthesis, creates a grid layout with the number of rows and columns specified by the arguments. Row 2: Class, Grid Layout; Constructor, Grid Layout, left parenthesis, i n t, number Of Rows, comma, i n t, number Of Columns, comma, i n t, h Gap, comma, i n t, v Gap, right parenthesis, creates a grid layout with the specified number of rows and columns and with a horizontal gap of h Gap pixels between columns and a vertical gap of v Gap pixels between rows. Horizontal gaps are also placed at the left and right edges, and vertical gaps are placed at the top and bottom edges.

Back to Table

Code for the Chess Board class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Using Grid Layout to organize our window. Line 2 Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x, dot, swing, dot, asterisk, semicolon. Line 6: import, java, dot, a w t, dot, asterisk, semicolon. Line 7: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 8: Blank. Line 9: public, class, Chess Board, extends, J Frame. Line 10: Left brace. Line 11, indented once: private, Container, contents, semicolon. Line 12, indented once: private, J Button, left square brace, right square brace, left square brace, right square brace, squares, semicolon. Line 13, indented once: private, String, left square brace, right square brace, letters, equals. Line 14, indented a few times: Left brace, open quotes, a, close quotes, comma, open quotes, b, close quotes, comma, open quotes, c, close quotes, comma, open quotes, d, close quotes, comma, open quotes, e, close quotes, comma, open quotes, f, close quotes, open quotes, g, close quotes, comma, open quotes, h, close quotes, right brace, semicolon. Line 15, indented once: private, i n t, left square brace, right square brace, digits, equals, left brace, 1, comma, 2, comma, 3, comma, 4, comma, 5, comma, 6, comma, 7, comma, 8, right brace, semicolon. Line 16: Blank. Line 17, indented once: public, Chess Board, left parenthesis, right parenthesis. Line 18, indented once: Left brace. Line 19, indented twice: super, left parenthesis, open quotes, Click a square to reveal its position, close quotes, right parenthesis, semicolon. Line 20, indented twice: contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 21: Blank. Line 22, indented twice: forward slash, forward slash, set layout to an 8-by-8 Grid. Line 23, indented twice: contents, dot, set Layout, left parenthesis, new, Grid Layout, left parenthesis, letters, dot, length, comma, digits, dot, length, right parenthesis, right parenthesis, semicolon. Line 24: Blank. Line 25, indented twice: squares, equals, new, J Button, left square brace, letters, dot, length, right square brace, left square brace, digits, dot, length, right square brace, semicolon. Line 26: Blank. Line 27, indented twice: Button Handler, b h, equals, new, Button Handler, parenthesis, right parenthesis, semicolon. Line 28: Blank. Line 29, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, letters, dot, length, semicolon, i, plus, plus, right parenthesis. Line 30, indented twice: Left brace. Line 31, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, digits, dot, length, semicolon, j, plus, plus, right parenthesis. Line 32, indented 3 times: Left brace. 33, indented 4 times: forward slash, forward slash, instantiate J Button array. Line 34, indented 4 times: squares, left square brace, i, right square brace, left square brace, j, right square brace, equals, new, J Button, left parenthesis, right parenthesis, semicolon. Line 35: Blank. Line 36, indented 4 times: forward slash, forward slash, make every other square red. Line 37, indented 4 times: if, left parenthesis, left parenthesis, i, plus, j, right parenthesis, percentage, 2, equals, equals, 0, right parenthesis. Line 38, indented 5 times: squares, left square brace, i, right square brace, left square brace, j, right square brace, dot, set Background, left parenthesis, Color, dot, RED, right parenthesis, semicolon. Line 39: Blank. Line 40, indented 4 times: forward slash, forward slash, add the J Button. Line 41, indented 4 times: contents, dot, add, left parenthesis, squares, left square brace, i, right square brace, left square brace, j, right square brace, right parenthesis, semicolon. Line 42: Blank. Line 43, indented 4 times: forward slash, forward slash, register listener on button. Line 44, indented 4 times: squares, left square brace, i, right square brace, left square brace, j, right square brace, add Action Listener, left parenthesis, b h, right square brace, semicolon. Line 45, indented 3 times: right brace. Line 46, indented twice: Right brace. Line 47: Blank. Line 48, indented twice: set Size, left parenthesis, 400, comma, 400, right parenthesis, semicolon. Line 49, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 50, indented once: Right brace. Line 51: Blank. Line 52, indented once: private, class, Button Handler, implements, Action Listener. Line 53, indented once: left brace. Line 54, indented twice: public, void, action Performed, left parenthesis, Action Event, a e, right parenthesis. Line 55, indented twice: Left brace. Line 56, indented 3 times: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, letters, dot, length, semicolon, i, plus, plus, right parenthesis. Line 57, indented 3 times: Left brace. Line 58, indented 4 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, digits, dot, length, semicolon, j, plus, plus, right parenthesis. Line 59, indented 4 times: Left brace. Line 60, indented 5 times: if, left parenthesis, a e, dot, get Source, left parenthesis, right parenthesis, equals, equals, squares, left square brace, i, right square brace, left square brace, j, right square brace, right parenthesis. Line 61, indented 5 times: Left brace. Line 62, indented 6 times: squares, left square brace, i, right square brace, left square brace, j, right square brace, dot, set Text, left parenthesis, letters, left square brace, i, right square brace, plus, digits, left square brace, j, right square brace, right parenthesis, semicolon. Line 63, indented 6 times: return, semicolon. Line 64, indented 5 times: Right brace. Line 65, indented 4 times: Right brace. Line 66, indented 3 times: Right brace. Line 67, indented twice: Right brace. Line 68, indented once: Right brace. Line 69: Blank. Line 70 public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 71, indented once: Left brace. Line 72, indented twice: Chess Board, my Game, equals, new, Chess Board, left parenthesis, right parenthesis, semicolon. Line 73, indented twice: my Game, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 74, indented once: Right brace. Line 75: Right brace.

Back to Example

Code for the Tile Puzzle class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Tile Puzzle class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: public, class, Tile Puzzle. Line 6: Left brace. Line 7, indented once: private, String, left square brace, right square brace, left square brace, right square brace, tiles, semicolon. Line 8, indented once: private, i n t, side, semicolon, forward slash, forward slash, grid size. Line 9, indented once: private, i n t, empty Row, semicolon. Line 10, indented once: private, i n t, empty C o l, semicolon. Line 11: Blank.

Line 12, indented once: forward slash, asterisk, asterisk, constructor. Line 13, indented once: Asterisk, at, param, new Side, grid size. Line 14, indented once: Asterisk, forward slash. Line 15, indented once: public, Tile Puzzle, left parenthesis, i n t, new Side, right parenthesis. Line 16, indented once: Left brace. Line 17, indented twice: set Up Game, left parenthesis, new Side, right parenthesis, semicolon. Line 18, indented once: right brace. Line 19: Blank. Line 20, indented once: forward slash, asterisk, asterisk, set Up Game. Line 21, indented once: Asterisk, at, param, new Side, grid size. Line 22, indented once: Asterisk, forward slash. Line 23, indented once: public, void, set Up Game, left parenthesis, i n t, new Side, right parenthesis. Line 24, indented once: Left brace. Line 25, indented twice: if, left parenthesis, new Side, less than, 1, right parenthesis. Line 26, indented 3 times: side, equals, 3, semicolon. Line 27, indented twice: else. Line 28, indented 3 times: side, equals, new Side, semicolon. Line 29, indented twice: empty Row, equals, side, minus, 1, semicolon. Line 30, indented once: empty C o l, equals, side, minus, 1, semicolon. Line 31, indented once: tiles, equals, new, String, left square brace, side, right square brace, left square brace, side, right square brace, semicolon. Line 32: Blank. Line 33, indented twice: forward slash, forward slash, initialize tiles. Line 34, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, side, semicolon, i, plus, plus, right parenthesis. Line 35, indented twice: Left brace. Line 36, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, side, semicolon, j, plus, plus, right parenthesis. Line 37, indented 3 times: Left brace. Line 38, indented 4 times: tiles, left square brace, i, right square brace, left square brace, j, right square brace, equals, String, dot, value Of, left parenthesis, left parenthesis, side, asterisk, side, right parenthesis. Line 39, indented a few times: minus, left parenthesis, side, asterisk, i, plus, j, plus, 1, right parenthesis, right parenthesis, semicolon. Line 40, indented 3 times: Right brace. Line 41, indented twice: Right brace. Line 42, indented twice: forward slash, forward slash, set empty cell label to blank. Line 43, indented twice: tiles, left square brace, side, minus, 1, right square brace, left square brace, side, minus, 1, right square brace, equals, open quotes, close quotes, semicolon. Line 44, indented once: Right brace. Line 45: Blank. Line 46, indented once: forward slash, asterisk, asterisk, get Side. Line 47, indented once: Asterisk, at, return side. Line 48, indented once: asterisk, forward slash. Line 49, indented once: public, i n t, get Side, left parenthesis, right parenthesis. Line 50, indented once: Left brace. Line 51, indented twice: return, side, semicolon. Line 52, indented once: right brace. Line 53: Blank. Line 54, indented once: forward slash, asterisk, asterisk, get Tiles. Line 55, indented once: asterisk, at, return a copy of tiles. Line 56, indented once: asterisk, forward slash. Line 57, indented once: public, String, left square brace, right square brace, left square brace, right square brace, get Tiles, left parenthesis, right parenthesis. Line 58, indented once:

Left brace. Line 59, indented twice: String, left square brace, right square brace, left square brace, right square brace, copy Of Tiles, equals, new, String, left square brace, side, right square brace, left square brace, side, right square brace, semicolon. Line 60: Blank. Line 61, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, side, semicolon, i, plus, plus, right parenthesis. Line 62, indented twice: Left brace. Line 63, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, side, semicolon, j, plus, plus, right parenthesis. Line 64, indented 3 times: left brace. Line 65, indented 4 times: copy Of Tiles, left square brace, i, right square brace, left square brace, j, right square brace, equals, tiles, left square brace, i, right square brace, left square brace, j, right square brace, semicolon. Line 66, indented 3 times: right brace. Line 67, indented twice: Right brace. Line 68, indented twice: return, copy Of Tiles, semicolon. Line 69, indented once: right brace. Line 70: Blank. Line 71, indented once: forward slash, asterisk, asterisk, try To Play. Line 72, indented once: asterisk, enable play if play is legal. Line 73, indented once: asterisk, at, return true if the play is legal, false otherwise. Line 74, indented once: asterisk, forward slash. Line 75, indented once: public, boolean, try To Play, left parenthesis, i n t, row, comma, i n t, c o l, right parenthesis. Line 76, indented once: Left brace. Line 77, indented twice: if, left parenthesis, possible To Play, left parenthesis, row, comma, c o l, right parenthesis, right parenthesis. Line 78, indented twice: left brace. Line 79, indented 3 times: forward slash, forward slash, play, colon, switch empty String and tile label at row, comma, c o l. Line 80, indented 3 times: tiles, left square brace, empty Row, right square brace, left square brace, empty C o l, right square brace, equals, tiles, left square brace, row, right square brace, left square brace, c o l, right square brace, semicolon. Line 81, indented 3 times: tiles, left square brace, row, right square brace, left square brace, c o l, right square brace, equals, open quotes, close quotes, semicolon. Line 82, indented 3 times: forward slash, forward slash, update empty Row and empty C o l. Line 83, indented 3 times: empty Row, equals, row, semicolon. Line 84, indented 3 times: empty C o l, equals, c o l, semicolon. Line 85, indented 3 times: return, true, semicolon. Line 86, indented twice: right brace. Line 87, indented twice: else. Line 88, indented 3 times: return, false, semicolon. Line 89, indented once: right brace. Line 90: Blank. Line 91, indented once: forward slash, asterisk, asterisk, possible To Play. Line 92, indented once: Asterisk, at, return true if the play is legal, comma, false otherwise. Line 93, indented once: Asterisk, forward slash. Line 94, indented once: public, boolean, possible To Play, left parenthesis, i n t, row, comma, i n t, c o l, right parenthesis. Line 95, indented once: Left brace. Line 96, indented twice: if, left parenthesis, left parenthesis, c o l, equals, equals, empty C o l, ampersand, ampersand, Math, dot, a b s, left parenthesis, row, minus, empty Row, right parenthesis, equals, equals, 1, right parenthesis. Line 97, indented a few times: pipe, pipe, left parenthesis, row, equals, equals, empty Row, ampersand, ampersand, Math, dot, a b s, left parenthesis, c o l, minus, empty C o l, right parenthesis, equals, equals, 1, right parenthesis, right parenthesis. Line 98, indented 3 times: return, true, semicolon. Line 99, indented twice: else. Line 100, indented 3 times: return, false, semicolon. Line 101, indented once: right brace. Line 102: Blank. Line 103, indented once: forward slash, asterisk, asterisk, won. Line 104, indented once: Asterisk, at, return true if the tiles are all at the right place, comma, false otherwise. Line 105, indented once: Asterisk, forward slash. Line 106, indented once: public, boolean, won, left parenthesis, right parenthesis. Line 107, indented once: Left brace. Line 108, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, side, semicolon, i, plus, plus, right parenthesis. Line 109, indented twice: Left brace. Line 110, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, side, semicolon, j, plus, plus, right parenthesis. Line 111, indented 3 times: Left brace. Line 112, indented 4 times: if, left parenthesis, exclamation mark, left parenthesis, tiles, left square brace, i, right square brace, left square brace, j, right square brace, dot, equals, left parenthesis. Line 113, indented a few times: String, dot, value Of, left parenthesis, i, asterisk, side, plus, j, plus, 1, right parenthesis, right parenthesis, right parenthesis. Line 114, indented a few times: ampersand, ampersand, left parenthesis, i, not equals, side, minus, 1, pipe, pipe, j, not equals, side, minus, 1, right parenthesis, right parenthesis. Line 115, indented 5 times: return, false, semicolon. Line 116, indented 3 times: Right brace. Line 117, indented twice: Right brace. Line 118, indented twice: return, true, semicolon. Line 119, indented once: Right brace. Line 120: Right brace.

Back to Example

Code for the Grid Button class. Program code. In the code, the words in the variable names are merged. Line 1: forward slash, asterisk, asterisk, Grid Button class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x, dot, swing, dot, J Button, semicolon. Line 6: Blank. Line 7: public, class, Grid Button, extends, J Button. Line 8: Left brace. Line 9, indented once: private, i n t, row, semicolon. Line 10, indented once: private, i n t, column, semicolon. Line 11: Blank. Line 12, indented once: forward slash, asterisk, asterisk, Constructor. Line 13, indented once: Asterisk, at, param, title. Line 14, indented once: Asterisk, at, param, new Row, row. Line 15, indented once: Asterisk, at, param, new Column column. Line 16, indented once: forward slash, asterisk. Line 17, indented once: public, Grid Button, left parenthesis, String, title, comma, i n t, new Row, comma, i n t, new Column, right parenthesis. Line 18, indented once: Left brace. Line 19, indented twice: super, left parenthesis, title, right parenthesis, semicolon. Line 20, indented twice: set Row, left parenthesis, new Row, right parenthesis, semicolon. Line 21, indented once: set Column, left parenthesis, new Column, right parenthesis, semicolon. Line 22, indented once: Right brace. Line 23: Blank. Line 24, indented once: forward slash, asterisk, asterisk, get Row method, comma, accessor for row. Line 25, indented once: Asterisk, at, return row. Line 26, indented once: Asterisk, forward slash. Line 27, indented once: public, i n t, get Row, left parenthesis, right parenthesis. Line 28, indented once: Left brace. Line 29, indented twice: return, row, semicolon. Line 30, indented once: right brace. Line 31: Blank. Line 32, indented once: forward slash, asterisk, asterisk, get Column method, comma, accessor for column. Line 33, indented once: asterisk, at, return column. Line 34: Asterisk, forward slash. Line 35, indented once: public, i n t, get Column, left parenthesis, right parenthesis. Line 36, indented once: Left brace. Line 37, indented twice: return, column, semicolon. Line 38, indented once: Right brace. Line 39: Blank. Line 40, indented once: forward slash, asterisk, asterisk, set Row method, comma, mutator for row. Line 41, indented once: asterisk, at, param new Row, comma, an i n t. Line 42, indented once: asterisk, forward slash. Line 43, indented once: public, void, set Row, left parenthesis, i n t, new Row, right parenthesis. Line 44, indented once: left brace. Line 45, indented twice: if, left parenthesis, new Row, greater than or equals, 0, right parenthesis. Line 46, indented 3 times: row, equals, new Row, semicolon. Line 47, indented once: Right brace. Line 48: Blank. Line 49, indented once: forward slash, asterisk, asterisk, set

Column method. Line 50, indented once: asterisk, at, param, new Column, comma, an i n t. Line 51, indented once: Asterisk, forward slash. Line 52, indented once: public, void, set Column, left parenthesis, i n t, new Column, right parenthesis. Line 53, indented once: Left brace. Line 54, indented twice: if, left parenthesis, new Column, greater than or equals, 0, right parenthesis. Line 55, indented 3 times, equals, new Column, semicolon. Line 56, indented once: Right brace. Line 57: Right brace.

Back to Example

Code for the Tile Puzzle View Controller class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Using Grid Layout dynamically. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x, dot, swing, dot, asterisk, semicolon. Line 6: import, java, dot, a w t, dot, asterisk, semicolon. Line 7: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 8: import, java, dot, util, dot, Random, semicolon. Line 9: Blank. Line 10: public, class, Tile Puzzle View Controller, extends, J Frame. Line 11: Left brace. Line 12, indented once: private, Container, contents, semicolon. Line 13: Blank. Line 14, indented once: private, Grid Button, left square brace, right square brace, left square brace, right square brace, squares, semicolon. Line 15, indented once: private, Tile Puzzle, game, semicolon, forward slash, forward slash, the tile puzzle game. Line 16: Blank. Line 17, indented once:

public, Tile Puzzle View Controller, left parenthesis, Tile Puzzle, new Game, right parenthesis. Line 18, indented once: Left brace. Line 19, indented twice: super, left parenthesis, open quotes, The Tile Puzzle Game, close quotes, right parenthesis, semicolon. Line 20, indented twice: game, equals, new Game, semicolon. Line 21, indented twice: set Up Game G U I, left parenthesis, right parenthesis, semicolon. Line 22, indented once: right brace. Line 23: Blank. Line 24, indented once: public, void, set Up Game G U I, left parenthesis, right parenthesis. Line 25, indented once: Left brace. Line 26, indented twice: contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 27, indented twice: contents, dot, remove All, left parenthesis, right parenthesis, semicolon, forward slash, forward slash, remove all components. Line 28, indented once: contents, dot, set Layout, left parenthesis, new, Grid Layout, left parenthesis, game, dot, get Side, left parenthesis, right parenthesis, semicolon. Line 29, indented a few times: game, dot, get Side, left parenthesis, right parenthesis, right parenthesis, right parenthesis, semicolon. Line 30, indented twice: squares, equals, new, Grid Button, left square brace, game, dot, get Side, left parenthesis, right parenthesis, right square brace, left square grace, game, dot, get Side, left parenthesis, right parenthesis, right square brace, semicolon. Line 31: Blank. Line 32, indented twice: Button Handler, b h, equals, new, Button Handler, left parenthesis, right parenthesis, semicolon. Line 33: Blank. Line 34, indented twice: forward slash, forward slash, for each button, colon, instantiate button with appropriate button label, comma. Line 35, indented twice: forward slash, forward slash, add to container, comma, and register listener. Line 36, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, game, dot, get Side, left parenthesis, right parenthesis, semicolon, i, plus, plus, right parenthesis. Line 37, indented twice: Left brace. Line 38, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, game, dot, get Side, left parenthesis, right parenthesis, semicolon, j, plus, plus, right parenthesis. Line 39, indented 3 times: Left brace. Line 40, indented 4 times: squares, left square brace, i, right square brace, left square brace, j, right square brace, equals, new, Grid Button, left parenthesis, game, dot, get Tiles, left parenthesis, right parenthesis, left square brace, i, right square brace, left square brace, j, right square brace, comma, i, comma, j, right parenthesis, semicolon. Line 41, indented 4 times: contents, dot, add, left parenthesis, squares, left square brace, i, right square brace, left square brace, j, right square brace, right parenthesis, semicolon. Line 42, indented 4 times: squares, left square brace, i, right square brace, left square brace, j, right square brace, dot, add Action Listener, left b h, right parenthesis, semicolon. Line 43, indented 3 times: Right brace. Line 44, indented twice: Right brace. Line 45: Blank. Line 46, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 47, indented once: Right brace. Line 48: Blank. Line 49, indented once: public, void, update, left parenthesis, right parenthesis. Line 50, indented once: left brace. Line 51, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, game, dot, get Side, left parenthesis, right parenthesis, semicolon, i, plus, plus, right parenthesis. Line 52, indented twice: Left brace. Line 53, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, game, dot, get Side, left parenthesis, right parenthesis, semicolon, j, plus, plus, right parenthesis. Line 54, indented 3 times: Left brace. Line 55, indented 4 times: squares, left square brace, i, right square brace, left square brace, j, right square brace, dot, set Text, left parenthesis, game, dot, get Tiles, left parenthesis, right parenthesis, left square brace, i, right square brace, left square brace, j, right square brace, right parenthesis, semicolon. Line 56, indented 3 times: Right brace. Line 57, indented twice: right brace. Line 58: Blank. Line 59, indented twice: if, left parenthesis, game, dot, won, left parenthesis, right parenthesis, right parenthesis. Line 60, indented twice: Left brace. Line 61, indented 3 times: J Option Pane, dot, show Message Dialog, left parenthesis, Tile Puzzle View Controller, dot, this, comma. Line 62, indented a few times: close quotes, Congratulations, exclamation mark, You won, exclamation mark, back slash, n, Setting up new game, close quotes, right parenthesis, semicolon. Line 63, indented 3 times: Random, random, equals, new, Random, left parenthesis, right parenthesis, semicolon. Line 64, indented 3 times: i n t, side Of Puzzle, equals, 3, plus, random, dot, next I n t, left parenthesis, 4, right parenthesis, semicolon. Line 65, indented 3 times: game, dot, set Up Game, left parenthesis, side Of Puzzle, right parenthesis, semicolon. Line 66, indented 3 times: set Up Game G U I, left parenthesis, right parenthesis, semicolon. Line 67, indented twice: right brace. Line 68, indented once: right brace. Line 69: Blank. Line 70, indented once: private, class, Button Handler, implements, Action Listener. Line 71, indented once: left brace. Line 72, indented twice: public, void, action Performed, left parenthesis, Action Event, a e, right parenthesis. Line 73, indented twice: Left brace. Line 74, indented 3 times: Grid Button, button, equals, left parenthesis, Grid Button, right parenthesis, a e, dot, get Source, left parenthesis, right parenthesis, semicolon. Line 75, indented 3 times: if, left parenthesis, game, dot, try To Play, left parenthesis, button, dot, get Row, left parenthesis, right parenthesis, comma, button, dot, get Column, left parenthesis, right parenthesis, right parenthesis, right parenthesis. Line 76, indented 4 times: update, left parenthesis, right parenthesis, semicolon. Line 77, indented twice: Right brace. Line 78, indented once: Right brace. Line 79: Right brace.

Back to Example

Code for the Play Tile Puzzle class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Play Tile Puzzle class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5:

import, java x, dot, swing, dot, J Frame, semicolon. Line 6: Blank. Line 7: public, class, Play Tile Puzzle, extends, J Frame. Line 8: Left brace. Line 9, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 10, indented once: Left brace. Line 11, indented twice: Tile Puzzle, puzzle, equals, new, Tile Puzzle, left parenthesis, 3, right parenthesis, semicolon. Line 12, indented twice: Tile Puzzle View Controller, puzzle G u i. Line 13, indented 3 times: equals, new, Tile Puzzle View Controller, left parenthesis, puzzle, right parenthesis, semicolon. Line 14, indented twice: puzzle G u i, dot, set Size, left parenthesis, 300, comma, 300, right parenthesis, semicolon. Line 15, indented twice: puzzle G u i, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 16, indented once: Right brace. Line 17: Right brace.

Back to Example

A table detailing the Container add method for using a Border Layout. The table has the following details. Return value: void. Method name and argument list: add, left parenthesis, Component, c, comma, Object, border layout Area, right parenthesis, adds the component c to the container. The area defined by border layout Area can be specified using any of the following static String constants in the Border Layout class: NORTH, SOUTH, EAST, WEST, CENTER.

Back to Table

A table detailing Border Layout constructors. The table has 2 rows and 2 columns. It has the following column headings. Class. Constructor. The row details are the following. Row 1: Class, Border Layout; Constructor, Border Layout, left parenthesis, right parenthesis, creates a new border layout with no gaps between components. Row 2: Class, Border Layout; Constructor, Border Layout, left parenthesis, i n t, h Gap, comma, i n t, v Gap, right parenthesis, creates a border layout with a horizontal gap of h Gap pixels between components and a vertical gap of v Gap pixels between components. Unlike Grid Layout, horizontal gaps are not placed at the left and right edges, and vertical gaps are not placed at the top and bottom edges.

Back to Table

Code for the R G B Color Mixer class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, R G B Color Mixer class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java, dot, a w t, dot, Color, semicolon. Line 6: Blank. Line 7: public, class, R G B Color Mixer. Line 8: Left brace. Line 9, indented once: public, static, i n t, POSITION underscore CHANGE, equals, 0, semicolon. Line 10, indented once: public, static, i n t, POSITION underscore RED, equals, 1, semicolon. Line 11, indented once: public, static, i n t, POSITION underscore GREEN, equals, 2, semicolon. Line 12, indented once: public, static, i n t, POSITION underscore BLUE, equals, 3, semicolon. Line 13: Blank. Line 14, indented once: private, i n t, delta, semicolon, forward slash, forward slash, color increment. Line 15, indented once: private, Color, color, semicolon, forward slash, mixed color. Line 16: Blank. Line 17, indented once: forward slash, asterisk, asterisk, Constructor. Line 18, indented once: Asterisk, at, param, new Delta, delta. Line 19, indented once: asterisk, forward slash. Line 20, indented once: public, R G B Color Mixer, left parenthesis, i n t, new Delta, right parenthesis. Line 21, indented once: Left brace. Line 22, indented twice: if, left parenthesis, new Delta, less than, minus 255, pipe, pipe, new Delta, greater than, 255, pipe, pipe, new Delta, equals, equals, 0, right parenthesis. Line 23, indented 3 times: delta, equals, 1, semicolon. Line 24, indented twice: else. Line 25, indented 3 times: delta, equals, new Delta, semicolon. Line 26, indented twice: if, left parenthesis, delta, greater than, 0, right parenthesis. Line 27, indented 3 times: color, equals, Color, dot, BLACK, semicolon. Line 28, indented twice: else. Line 29, indented 3 times: color, equals, Color, dot, WHITE, semicolon. Line 30, indented once: Right brace. Line 31: Blank. Line 32, indented once: forward slash, asterisk, asterisk, get Color method. Line 33, indented once: asterisk, at, return, color. Line 34, indented once: asterisk, forward slash. Line 35, indented once: public, Color, get Color, left parenthesis, right parenthesis. Line 36, indented once: Left brace. Line 37, indented twice: return, color, semicolon. Line 38, indented once: right brace. Line 39: Blank. Line 40, indented once: forward slash, asterisk, asterisk, invert Delta method. Line 41, indented once: asterisk, sets delta to its opposite. Line 42, indented once: asterisk, forward slash. Line 43, indented once: public, void, invert Delta, left parenthesis, right parenthesis. Line 44, indented once: Left brace. Line 45, indented twice: delta, equals, minus, delta, semicolon. Line 46, indented once: right brace. Line 47: Blank. Line 48, indented once: forward slash, asterisk, asterisk, add To Red method. Line 49, indented once: asterisk, add delta to red component of color. Line 50, indented once: Asterisk, forward slash. Line 51, indented once: public, void, add To Red, left parenthesis, right parenthesis. Line 52, indented once: Left brace. Line 53, indented twice: if, left parenthesis, color, dot, get Red, left parenthesis, right parenthesis, plus, delta, greater than, 255, right parenthesis. Line 54, indented 3 times: color, equals, new, Color, left parenthesis, 255, comma, color, dot, get Green, left parenthesis, right parenthesis, comma, color, dot, get Blue, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 55, indented twice: else, if, left parenthesis, color, dot, get Red, left parenthesis, right parenthesis, plus, delta, less than, 0, right parenthesis. Line 56, indented 3 times: color, equals, new, Color, left parenthesis, 0, comma, color, dot, get Green, left parenthesis, right parenthesis, comma, color, dot, get Blue, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 57, indented twice: else. Line 58, indented 3 times: color, equals, new, Color, left parenthesis, color, dot, get Red, left parenthesis, right parenthesis, plus, delta, comma, color, dot, get Green, left parenthesis, right parenthesis, comma. Line 59, indented a few times: color, dot, get Blue, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 60, indented once: right brace. Line 61: Blank. Line 62, indented once: forward slash, asterisk, asterisk, add To Green method. Line 63, indented once: asterisk, add delta to green component of color. Line 64, indented once: Asterisk, forward slash. Line 65, indented once: public, void, add To Green, left parenthesis, right parenthesis. Line 66, indented once: Left brace. Line 67, indented twice: if, left parenthesis, color, dot, get Green, left parenthesis, right parenthesis, plus, delta, greater than, 255, right parenthesis. Line 68, indented 3 times: color, equals, new, Color, left parenthesis, color, dot, get Red, left parenthesis, right parenthesis, comma, 255, comma, color, dot, get Blue, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 69, indented twice: else, if, left parenthesis, color, dot, get Green, left parenthesis, right parenthesis, plus, delta, less than, 0, right parenthesis. Line 70, indented 3 times: color, equals, new, Color, left parenthesis, color, dot, get Red, left parenthesis, right parenthesis, comma, 0, comma, color, dot, get Blue, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 71, indented twice: else. Line 72, indented 3 times: color, equals, new, Color, left parenthesis, color, dot, get Red, left parenthesis, right parenthesis, comma, color, dot, get Green, left parenthesis, right parenthesis, plus, delta, comma. Line 73, indented a few times: color, dot, get Blue, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 74, indented once: right brace. Line 75: Blank. Line 76, indented once: forward slash, asterisk, asterisk, add To Blue method. Line 77, indented once: asterisk, add delta to blue component of color. Line 78, indented once: asterisk, forward slash. Line 79, indented once: public, void, add To Blue, left parenthesis, right parenthesis. Line 80, indented once: Left brace. Line 81, indented twice: if, left parenthesis, color, dot, get Blue, left parenthesis, right parenthesis, plus, delta, greater than, 255, right parenthesis. Line 82, indented 3 times: color, equals, new, Color, left parenthesis, color, dot, get Red, left parenthesis, right parenthesis, comma, color, dot, get Green, left parenthesis, right parenthesis, comma, 255, right parenthesis, semicolon. Line 83, indented twice: else, if, left parenthesis, color, dot, get Blue, left parenthesis, right parenthesis, plus, delta, less than, 0, right parenthesis. Line 84, indented 3 times: color, equals, new, Color, left parenthesis, color, dot, get Red, left parenthesis, right parenthesis, comma, color, dot, get Green, left parenthesis, right parenthesis, comma, 0, right parenthesis, semicolon. Line 85, indented twice: else. Line 86, indented 3 times: color, equals, new, Color, left parenthesis, color, dot, get Red, left parenthesis, right parenthesis, comma, color, dot, get Green, left parenthesis, right parenthesis, comma. Line 87, indented a few times: color, dot, get Blue, left parenthesis, right parenthesis, plus, delta, right parenthesis, semicolon. Line 88, indented once: right brace. Line 89: Blank. Line 90, indented once: forward slash, asterisk, asterisk, get Method Label method. Line 91, indented once: Asterisk, at, param, position 0, colon, Change, comma, 1, colon, Red, comma, 2, colon, Green, comma, 3, colon, Blue. Line 92, indented once: asterisk, at, return a String to use as the text to label. Line 93, indented once: Asterisk, the calling of one of the methods. Line 94, indented once: Asterisk, invert Delta, comma, add To Red, comma, add To Green, comma, and add To Blue. Line 95, indented once: Asterisk, forward slash. Line 96, indented once: public, String, get Method Label, left parenthesis, i n t, position, right parenthesis. Line 97, indented once: left brace. Line 98, indented twice: if, left parenthesis, position, equals, equals, POSITION underscore CHANGE, right parenthesis. Line 99, indented 3 times: return, open quotes, CHANGE DIRECTION, close quotes, semicolon. Line 100, indented twice: else, if, left parenthesis, position, equals, equals, POSITION underscore RED, right parenthesis. Line 101, indented 3 times: return, open quotes, RED, close quotes, semicolon. Line 102, indented twice: else, if, left parenthesis, position, equals, equals, POSITION underscore GREEN, right parenthesis. Line 103, indented 3 times: return, open quotes, GREEN, close quotes, semicolon. Line 104, indented twice: else, if, left parenthesis, position, equals, equals, POSITION underscore BLUE, right parenthesis. Line 105, indented 3 times: return, open quotes, BLUE, close quotes, semicolon. Line 106, indented twice: else. Line 107, indented 3 times: return, open quotes, close quotes, semicolon. Line 108, indented once: right brace. Line 109: Blank. Line 110, indented once: forward slash, asterisk, asterisk, get Title method. Line 111, indented once: Asterisk, at, return a String to use as title for this object. Line 112, indented once: Asterisk, forward slash. Line 113, indented once: public, String, get Title, left parenthesis, right parenthesis. Line 114, indented once: left brace. Line 115, indented 3 times: return, open quotes, Start Mixing Colors, close quotes, semicolon. Line 116, indented once: right brace. Line 117: Blank. Line 118, indented once: forward slash, asterisk, asterisk, to String method. Line 119, indented once: Asterisk, at, return a String representation of color. Line 120, indented once: asterisk, forward slash. Line 121, indented once: public, String, to String, left parenthesis, right parenthesis. Line 122, indented once: left brace. Line 123, indented twice: String, color Text, equals, color, dot, to String, left parenthesis, right parenthesis, semicolon. Line 124, indented twice: i n t, bracket Index, equals, color Text, dot, index Of, left parenthesis, open quotes, left square brace, close quotes, right parenthesis, semicolon. Line 125, indented once: if, left parenthesis, bracket Index, greater than or equals, 0, right parenthesis. Line 126, indented 3 times: color Text, equals, color Text, dot, sub string, left parenthesis, bracket Index, comma, color Text, dot, length, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 127, indented twice: return, color Text, semicolon. Line 128, indented twice: right brace. Line 129: right brace.

Back to Example

Code for the Border Layout View class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Border Layout View class. Line 2: Asterisk, Reusable generic frame using a Border Layout. Line 3: Asterisk, Four buttons occupy the North, East, South, and West positions. Line 4: Asterisk, A label with centered text occupies the center position. Line 5: Asterisk, Accessors are provided so that a Controller can access. Line 6: Asterisk, the four buttons and the label. Line 7: Asterisk, Anderson, comma, Franceschi. Line 8: Asterisk, forward slash. Line 9: Blank. Line 10: import, java x, dot, swing, dot, asterisk, semicolon. Line 11: import, java, dot, a w t, dot, asterisk, semicolon. Line 12: Blank. Line 13: public, class, Border Layout View, extends, J Frame. Line 14: Left brace. Line 15, indented once: private, J Button, north, comma, east, comma, south, comma, west, semicolon. Line 16, indented once: private, J Label, center, semicolon. Line 17: Blank. Line 18, indented once: public, Border Layout View, left parenthesis, right parenthesis. Line 19, indented once: Left brace. Line 20, indented twice: super, left parenthesis, right parenthesis, semicolon. Line 21, indented twice: Container, contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 22, indented twice: contents, dot, set Layout, left parenthesis, new, Border Layout, left parenthesis, right parenthesis, right parenthesis, semicolon, forward slash, forward slash, optional. Line 23: Blank. Line 24: forward slash, forward slash, instantiate button objects. Line 25, indented twice: north, equals, new, J Button, left parenthesis, right parenthesis, semicolon. Line 26, indented twice: east, equals, new, J Button, left parenthesis, right parenthesis, semicolon. Line 27, indented twice: south, equals, new, J Button, left parenthesis, right parenthesis, semicolon. Line 28, indented twice: west, equals, new, J Button, left parenthesis, right parenthesis, semicolon. Line 29: Blank. Line 30, indented twice: center, equals, new, J Label, left parenthesis, open quotes, close quotes, comma, Swing Constants, dot, CENTER, right parenthesis, semicolon. Line 31, indented twice: center, dot, set Opaque, left parenthesis, true, right parenthesis, semicolon. Line 32: Blank. Line 33, indented twice: forward slash, forward slash, order of adding components is not important. Line 34, indented twice: contents, dot, add, left parenthesis, north, comma, Border Layout, dot, NORTH, right parenthesis, semicolon. Line 35, indented twice: contents, dot, add, left parenthesis, east, comma, Border Layout, dot, EAST, right parenthesis, semicolon. Line 36, indented twice: contents, dot, add, left parenthesis, south, comma, Border Layout, dot, SOUTH, right parenthesis, semicolon. Line 37, indented twice: contents, dot, add, left parenthesis, west, comma, Border Layout, dot, WEST, right parenthesis, semicolon. Line 38, indented twice: contents, dot, add, left parenthesis, center, comma, Border Layout, dot, CENTER, right parenthesis, semicolon. Line 39: Blank. Line 40, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 41, indented once: right brace. Line 42: Blank. Line 43, indented once: public, J Button, get North, left parenthesis, right parenthesis. Line 44, indented once: left brace. Line 45, indented twice: return, north, semicolon. Line 46, indented once: right brace. Line 47: Blank. Line 48, indented once: public, J Button, get East, left parenthesis, right parenthesis. Line 49: Left brace. Line 50, indented twice: return, east, semicolon. Line 51, indented once: right brace. Line 52: Blank. Line 53, indented once: public, J Button, get South, left parenthesis, right parenthesis. Line 54, indented once: Left brace. Line 55, indented twice: return, south, semicolon. Line 56, indented once: right brace. Line 57: Blank. Line 58, indented once: public, J Button, get West, left parenthesis, right parenthesis. Line 59, indented once: Left brace. Line 60, indented twice: return, west, semicolon. Line 61, indented once: right brace. Line 62: Blank. Line 63, indented once: public, J Label, get Center, left parenthesis, right parenthesis. Line 64, indented once: Left brace. Line 65, indented twice: return, center, semicolon. Line 66, indented once: right brace. Line 67: right brace.

Back to Example

Code for the R G B Color Mixer Controller class. Program code. In the code, the words in the variable names are merged. Line 1: forward slash, asterisk, asterisk, R G B Color Mixer Controller class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 6: import, java x, dot, swing, dot, asterisk, semicolon. Line 7: Blank. Line 8: public, class, R G B Color Mixer Controller, implements, Action Listener. Line 9: Left brace. Line 10, indented once: private, R G B Color Mixer, model, semicolon. Line 11, indented once: private, Border Layout View, view, semicolon. Line 12: Blank. Line 13, indented once: public, R G B Color Mixer Controller, left parenthesis, R G B Color Mixer, new Model, comma. Line 14, indented a few times: Border Layout View, new View, right parenthesis. Line 15, indented once: Left brace. Line 16, indented twice: model, equals, new Model, semicolon. Line 17, indented twice: view, equals, new View, semicolon. Line 18: Blank. Line 19, indented twice: view, dot, set Title, left parenthesis, model, dot, get Title, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 20: Blank. Line 21, indented twice: J Button, north, equals, view, dot, get North, left parenthesis, right parenthesis, comma, west, equals, view, dot, get West, left parenthesis, right parenthesis, comma. Line 22, indented a few times: south, equals, view, dot, get South, left parenthesis, right parenthesis, comma, east, equals, view, dot, get East, left parenthesis, right parenthesis, semicolon. Line 23: Blank. Line 24, indented twice: north, dot, set Text, left parenthesis, model, dot, get Method Label, left parenthesis, R G B Color Mixer, dot, POSITION underscore CHANGE, right parenthesis, right parenthesis, semicolon. Line 25, indented twice: west, dot, set Text, left parenthesis, model, dot, get Method Label, left parenthesis, R G B Color Mixer, dot, POSITION underscore RED, right parenthesis, right parenthesis, semicolon. Line 26, indented twice: south, dot, set Text, left parenthesis, model, dot, get Method Label, left parenthesis, R G B Color Mixer, dot, POSITION underscore GREEN, right parenthesis, right parenthesis, semicolon. Line 27, indented twice: east, dot, set Text, left parenthesis, model, dot, get Method Label, left parenthesis, R G B Color Mixer, dot, POSITION underscore BLUE, right parenthesis, right parenthesis, semicolon. Line 28: Blank. Line 29, indented twice: north, dot, add Action Listener, left parenthesis, this, right parenthesis, semicolon. Line 30, indented twice: west, dot, add Action Listener, left parenthesis, this, right parenthesis, semicolon. Line 31, indented twice: south, dot, add Action Listener, left parenthesis, this, right parenthesis, semicolon. Line 32, indented twice: east, dot, add Action Listener, left parenthesis, this, right parenthesis, semicolon. Line 33: Blank. Line 34, indented twice: update View, left parenthesis, right parenthesis, semicolon. Line 35, indented once: right brace. Line 36:

Blank. Line 37, indented once: public, void, action Performed, left parenthesis, Action Event, a e, right parenthesis. Line 38, indented once: Left brace. Line 39, indented twice: forward slash, forward slash, update the model. Line 40, indented twice: if, left parenthesis, a e, dot, get Source, left parenthesis, right parenthesis, equals, equals, view, dot, get North, left parenthesis, right parenthesis, right parenthesis. Line 41, indented 3 times: model, dot, invert Delta, left parenthesis, right parenthesis, semicolon. Line 42, indented twice: else, if, left parenthesis, a e, dot, get Source, left parenthesis, right parenthesis, equals, equals, view, dot, get West, left parenthesis, right parenthesis, right parenthesis. Line 43, indented 3 times: model, dot, add To Red, left parenthesis, right parenthesis, semicolon. Line 44, indented twice: else, if, left parenthesis, a e, dot, get Source, left parenthesis, right parenthesis, equals, equals, view, dot, get South, left parenthesis, right parenthesis, right parenthesis. Line 45, indented 3 times: model, dot, add To Green, left parenthesis, right parenthesis. Line 46, indented twice: else, if, left parenthesis, a e, dot, get Source, left parenthesis, right parenthesis, equals, equals, view, dot, get East, left parenthesis, right parenthesis, right parenthesis. Line 47, indented 3 times: model, dot, add To Blue, left parenthesis, right parenthesis, semicolon. Line 48, indented twice: forward slash, forward slash, update the view. Line 49, indented twice: update View, left parenthesis, right parenthesis, semicolon. Line 50, indented once: right brace. Line 51: Blank. Line 52, indented once: forward slash, asterisk, asterisk, update View method. Line 53, indented once: asterisk, updates the color and the text of the center label. Line 54, indented once: asterisk, forward slash. Line 55, indented once: public, void, update View, left parenthesis, right parenthesis. Line 56, indented once: left brace. Line 57, indented twice: J Label, label, equals, view, dot, get Center, left parenthesis, right parenthesis, semicolon. Line 58, indented twice: label, dot, set Background, left parenthesis, model, dot, get Color, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 59, indented twice: label, dot, set Text, left parenthesis, model, dot, to String, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 60, indented once: right brace. Line 61: right brace.

Back to Example

Code for the Mix Colors class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Mix Colors class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x, dot, swing, dot, J Frame, semicolon. Line 6: Blank. Line 7: public, class, Mix Colors. Line 8: Left brace. Line 9, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 10, indented once: Left brace. Line 11, indented twice: forward slash, forward slash, Instantiate the Model. Line 12, indented once: R G B Color Mixer color Mixer, equals, new, R G B Color Mixer, left parenthesis, minus 10, right parenthesis, semicolon. Line 13, indented twice: forward slash, forward slash, Instantiate the View. Line 14, indented twice: Border Layout View, color Mixer G u i, equals, new, Border Layout View, left parenthesis, right parenthesis, semicolon. Line 15, indented twice: color Mixer G u i, dot, set Size, left parenthesis, 350, comma, 250, right parenthesis, semicolon. Line 16, indented twice: color Mixer G u i, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 17, indented twice: forward slash, forward slash, Instantiate the Controller. Line 18, indented once: R G B Color Mixer Controller, controller. Line 19, indented 3 times: equals, new, R G B Color Mixer Controller, left parenthesis, color Mixer, comma, color Mixer G u i, right parenthesis, semicolon. Line 20, indented once: right brace. Line 21: right brace.

Back to Example

Code for the Color Grid Game interface. Program code. In the code, the words in the variable names are merged. Line 1: forward slash, forward slash, asterisk, Color Grid Game interface. Line 2, indented 3 times: Anderson, comma, Franceschi. Line 3, indented once: Asterisk, forward slash. Line 4: Blank. Line 5: import, java, dot, a w t, dot, Color, semicolon. Line 6: Blank. Line 7: public, interface, Color Grid Game. Line 8: Left brace. Line 9, indented once: public, String, get Title, left parenthesis, right parenthesis, semicolon. Line 10, indented once: public, i n t, get Size, left parenthesis, right parenthesis, semicolon. Line 11, indented once: public, i n t, get Number Of Colors, left parenthesis, right parenthesis, semicolon. Line 12, indented once: public, String, get Label, left parenthesis, i n t, index, right parenthesis, semicolon. Line 13, indented once: public, i n t, get Index, left parenthesis, String, label, right parenthesis, semicolon. Line 14, indented once: public, boolean, is Correct, left parenthesis, i n t, index, right parenthesis, semicolon. Line 15, indented once: public, Color, get Grid Color, left parenthesis, i n t, row, comma, i n t, col, right parenthesis, semicolon. Line 16: right brace.

Back to Example

Code for the Color Frequency Game class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Color Frequency Game class. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java, dot, util, dot, Random, semicolon. Line 6: import, java, dot, a w t, dot, Color, semicolon. Line 7: Blank. Line 8: public, class, Color Frequency Game, implements, Color Grid Game. Line 9: Left brace. Line 10, indented once: private, final, i n t, DEFAULT underscore SIZE, equals, 8, semicolon. Line 11, indented once: private, Color, left square brace, right square brace, left square brace, right square brace, grid Colors, semicolon.

Line 12, indented once: private, Color, left square brace, right square brace, coloring Colors, semicolon. Line 13, indented once: private, i n t, left square brace, right square brace, counts, semicolon, forward slash, forward slash, color frequencies in grid Colors. Line 14: Blank. Line 15, indented once: private, String, left square brace, right square brace, labels, semicolon. Line 16: Blank. Line 17, indented once: forward slash, asterisk, asterisk, Constructor. Line 18, indented once: Asterisk, at, param, size number of rows and columns in grid Colors. Line 19, indented once: Asterisk, @, param, colors coloring Colors. Line 20, indented once: Asterisk, at, param new Labels labels. Line 21, indented once: Asterisk, forward slash. Line 22, indented once: public, Color Frequency Game, left parenthesis, i n t, size, comma, Color, left square brace, right square brace, colors, comma. Line 23, indented a few times: String, left square brace, right square brace, new Labels, right parenthesis. Line 24, indented once: left brace. Line 25, indented twice: if, left parenthesis, size, less than or equals, 0, right parenthesis. Line 26, indented 3 times: size, equals, DEFAULT underscore SIZE, semicolon. Line 27, indented twice: grid Colors, equals, new, Color, left square brace, size, right square brace, left square brace, size, right square brace, semicolon. Line 28, indented twice: coloring Colors, equals, colors, semicolon. Line 29, indented twice: counts, equals, new, i n t, left square brace, coloring Colors, dot, length, right square brace, semicolon. Line 30, indented twice: fill Grid With Colors, left parenthesis, right parenthesis, semicolon. Line 31, indented twice: labels, equals, new Labels, semicolon. Line 32, indented once: right brace. Line 33: Blank. Line 34, indented once: forward slash, asterisk, asterisk, fill Grid With Colors method. Line 35, indented once: Asterisk, randomly fills grid Colors with colors from coloring Colors. Line 36, indented once: Asterisk, forward slash. Line 37, indented once: public, void, fill Grid With Colors, left parenthesis, right parenthesis. Line 38, indented once: Left brace. Line 39, indented twice: Random, random, equals, new, Random, left parenthesis, right parenthesis, semicolon. Line 40, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, grid Colors, dot, length, semicolon, i, plus, plus, right parenthesis. Line 41, indented once: Left brace. Line 42, indented a 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, grid Colors, left square brace, i, right square brace, dot, length, semicolon, j, plus, plus, right parenthesis. Line 43, indented 3 times: Left brace. Line 44, indented 4 times: i n t, color Index, equals, random, dot, next I n t, left parenthesis, coloring Colors, dot, length, right parenthesis, semicolon. Line 45, indented 4 times: grid Colors, left square brace, i, right square brace, left square brace, j, right square brace, equals, coloring Colors, left square brace, color Index, right square brace, semicolon. Line 46, indented 4 times: counts, left square brace, color Index, right square brace, plus, equals, 1, semicolon. Line 47, indented 3 times: right brace. Line 48, indented twice: right brace. Line 49, indented once: right brace. Line 50:

Blank. Line 51, indented once: forward slash, asterisk, asterisk, get Size method. Line 52, indented once: asterisk, at, return length of grid Colors. Line 53, indented once: asterisk, forward slash. Line 54, indented once: public, i n t, get Size, left parenthesis, right parenthesis. Line 55, indented once: left brace. Line 56, indented twice: return, grid Colors, dot, length, semicolon. Line 57, indented once: right brace. Line 58: Blank. Line 59, indented once: forward slash, asterisk, asterisk, get Number Of Colors method. Line 60, indented once: Asterisk, at, return length of coloring Colors. Line 61, indented once: asterisk, forward slash. Line 62, indented once: public, i n t, get Number Of Colors, lefrt parenthesis, right parenthesis. Line 63, indented once: Left brace. Line 64, indented twice: return, coloring Colors, dot, length, semicolon. Line 65, indented once: right brace. Line 66: Blank. Line 67, indented once: forward slash, asterisk, asterisk, get Grid Color method. Line 68, indented once: Asterisk, at, param row, comma, an i n t, comma, the row index. Line 69, indented once: Asterisk, at, param c o l, comma, an i n t, comma, the column index. Line 70, indented once: Asterisk, at, return grid Color, left square brace, row, right square brace, left square brace, c o l, right square brace. Line 71, indented once: asterisk, forward slash. Line 72, indented once: public, Color, get Grid Color, left parenthesis, i n t, row, comma, i n t, c o l, right parenthesis. Line 73, indented once: left brace. Line 74, indented twice: return, grid Colors, left square brace, row, right square brace, left square brace, c o l, right square brace, semicolon. Line 75, indented once: right brace. Line 76: Blank. Line 77, indented once: forward slash, asterisk, asterisk, highest Count method. Line 78, indented once: Asterisk, at, return the highest color frequency in the grid. Line 79, indented once: asterisk, forward slash. Line 80, indented once: public, i n t, highest Count, left parenthesis, right parenthesis. Line 81, indented once: Left brace. Line 82, indented twice: i n t, max, equals, counts, left square brace, 0, right square brace, semicolon. Line 83, indented twice: for, left parenthesis, i n t, i, equals, 1, semicolon, i, less than, counts, dot, length, semicolon, i, plus, plus, right parenthesis. Line 84, indented twice: left brace. Line 85, indented 3 times: if, left parenthesis, counts, left square brace, i, right square brace, greater than, max, right parenthesis. Line 86, indented 4 times: max, equals, counts, left square brace, i, right square brace, semicolon. Line 87, indented twice: right brace. Line 88, indented twice: return, max, semicolon. Line 89, indented once: right brace. Line 90: Blank. Line 91, indented once: forward slash, asterisk, asterisk, is Correct method. Line 92, indented once: Asterisk, at, param index, comma, an i n t. Line 93, indented once: Asterisk, at, return true if the frequency of the color for index. Line 94, indented once: Asterisk, is the highest, comma, false otherwise. Line 95, indented once: Asterisk, forward slash. Line 96, indented once: public, boolean, is Correct, left parenthesis, i n t, index, right parenthesis. Line 97, indented once: left brace. Line 98, indented twice: return, counts, left square brace, index, right square brace, equals, equals, highest Count, left parenthesis, right parenthesis, semicolon. Line 99, indented once: right brace. Line 100: Blank. Line 101, indented once: forward slash, asterisk, asterisk, get Title method. Line 102, indented once: Asterisk, at, return a String representing a title for this object. Line 103, indented once: Asterisk, forward slash. Line 104, indented once: public, String, get Title, left parenthesis, right parenthesis. Line 105, indented once: Left brace. Line 106, indented twice: return, open quotes, What is the most frequent color, question mark, close quotes, semicolon. Line 107, indented once: right brace. Line 108: Blank. Line 109, indented once: forward slash, asterisk, asterisk, get Label method. Line 110, indented once: Asterisk, at, param index, comma, an i n t. Line 111, indented once: Asterisk, at, return, labels, left square brace, index, right square brace. Line 112, indented once: Asterisk, forward slash. Line 113, indented once: public, String, get Label, left parenthesis, i n t, index, right parenthesis. Line 114, indented once: Left brace. Line 115, indented twice: return, labels, left square brace, index, right square brace, semicolon. Line 116, indented once: right brace. Line 117: Blank. Line 118, indented once: forward slash, asterisk, asterisk, get Index method. Line 119, indented once: Asterisk, at, param label, comma, a String. Line 120, indented once: Asterisk, at, return index of label in labels. Line 121, indented once: Asterisk, forward slash. Line 122, indented once: public, i n t, get Index, left parenthesis, String, label, right parenthesis. Line 123, indented once: Left brace. Line 124, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, labels, dot, length, semicolon, i, plus, plus, right parenthesis. Line 125, indented twice: Left brace. Line 126, indented 3 times: if, left parenthesis, labels, left square brace, i, right square brace, dot, equals, left parenthesis, label, right parenthesis, right parenthesis. Line 127, indented 4 times: return, i, semicolon. Line 128, indented twice: right brace. Line 129, indented twice: return, minus, 1, semicolon. Line 130, indented once: right brace. Line 131: right brace.

Back to Example

Code for the Bordered Label Grid View class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Bordered Label Grid View class. Line 2: Asterisk, Reusable generic frame using a Border Layout. Line 3: Asterisk, containing the west and center panels. Line 4: Asterisk, The west panel is made up of a vertical array of buttons. Line 5: Asterisk, The center panel is made up of a grid of labels. Line 6: Asterisk, Accessors are provided so that a Controller can access. Line 7: Asterisk, the array of buttons and the 2-dim array (grid) of labels. Line 8: Asterisk, Anderson, comma, Franceschi. Line 9: Asterisk, forward slash. Line 10: Blank. Line 11: import, java x, dot, swing, dot, asterisk, semicolon. Line 12: import, java, dot, a w t, dot, asterisk, semicolon. Line 13: import, java, dot, java, dot, a w t, dot, event, dot, Action Listener, semicolon. Line 14: Blank. Line 15: public, class, Bordered Label Grid View, extends, J Frame. Line 16: left brace. Line 17, indented once: private, J Panel, button Panel, semicolon, forward slash, forward slash, left, comma, holds array of buttons. Line 18, indented once: private, J Button, left square brace, right square brace, buttons, semicolon. Line 19, indented once: private, J Panel, grid, semicolon, forward slash, forward slash, center, comma, holds grid of labels. Line 20, indented once: private, J Label, left square brace, right square brace, left square brace, right square brace, labels, semicolon, forward slash, forward slash, grid of labels. Line 21: Blank. Line 22, indented once: public, Bordered Label Grid View, left parenthesis, i n t, number Of Buttons, comma, i n t, grid Size, right parenthesis. Line 23, indented once: left brace. Line 24, indented twice: super, left parenthesis, right parenthesis, semicolon. Line 25: Blank. Line 26, indented twice: Container, contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 27, indented twice: contents, dot, set Layout, left parenthesis, new, Border Layout, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 28: Blank. Line 29, indented twice: grid, equals, new, J Panel, left parenthesis, right parenthesis, semicolon. Line 30, indented twice: Grid Layout, game Grid, equals, new, Grid Layout, left parenthesis, grid Size, comma, grid Size, right parenthesis, semicolon. Line 31, indented twice: grid, dot, set Layout, left parenthesis, game Grid, right parenthesis, semicolon. Line 32: Blank.

Line 33, indented once: labels, equals, new, J Label, left square brace, grid Size, right square brace, left square brace, grid Size, right square brace, semicolon. Line 34, indented once: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, labels, dot, length, semicolon, i, plus, plus, right parenthesis. Line 35, indented twice: left brace. Line 36, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, labels, left square brace, i, right square brace, dot, length, semicolon, j, plus, plus, right parenthesis. Line 37, indented 3 times: Left brace. Line 38, indented 4 times: labels, left square brace, i, right square brace, left square brace, j, right square brace, equals, new, J Label, left parenthesis, right parenthesis, semicolon. Line 39, indented 4 times: labels, left square brace, i, right square brace, left square brace, j, right square brace, dot, set Opaque, left parenthesis, true, right parenthesis, semicolon. Line 40, indented 4 times: grid, dot, add, left parenthesis, labels, left square brace, i, right square brace, left square brace, j, right square brace, right parenthesis, semicolon. Line 41, indented 3 times: right brace. Line 42, indented twice: right brace. Line 43: Blank. Line 44, indented twice: button Panel, equals, new, J Panel, left parenthesis, right parenthesis, semicolon. Line 45, indented twice: Grid Layout, button Grid, equals, new, Grid Layout, left parenthesis, number Of Buttons, comma, 1, right parenthesis, semicolon. Line 46, indented twice: button Panel, dot, set Layout, left parenthesis, button Grid, right parenthesis, semicolon. Line 47: Blank. Line 48, indented twice: buttons, equals, new, J Button, left square brace, number Of Buttons, right square brace, semicolon. Line 49, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, buttons, dot, length, semicolon, i, plus, plus, right parenthesis. Line 50, indented twice: Left brace. Line 51, indented 3 times: buttons, left square brace, i, right square brace, equals, new, J Button, left parenthesis, right parenthesis, semicolon. Line 52, indented 3 times: button Panel, dot, add, left parenthesis, buttons, left square brace, i, right square brace, right parenthesis, semicolon. Line 53, indented twice: right brace. Line 54: Blank. Line 55, indented twice: contents, dot, add, left parenthesis, Border Layout, dot, WEST, comma, button Panel, right parenthesis, semicolon. Line 56, indented twice: contents, dot, add, left parenthesis, Border Layout, dot, CENTER, comma, grid, right parenthesis, semicolon. Line 57, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 58, indented once: right brace. Line 59: Blank. Line 60, indented once: public, void, set Button Text, left parenthesis, i n t, row, comma, String, text, right parenthesis. Line 61, indented once: left brace. Line 62, indented twice: buttons, left square brace, row, right square brace, dot, set Text, left parenthesis, text, right parenthesis, semicolon. Line 63, indented once: right brace. Line 64: Blank. Line 65, indented once: public, void, set Button Listener, left parenthesis, i n t, row, comma, Action Listener, a l, right parenthesis. Line 66, indented once: Left brace. Line 67, indented twice:

buttons, left square brace, row, right square brace, dot, add Action Listener, left parenthesis, a l, right parenthesis, semicolon. Line 68, indented once: right brace. Line 69: Blank. Line 70, indented once: public, void, set Label Background, left parenthesis, i n t, row, comma, i n t, c o l, comma, Color, color, right parenthesis. Line 71, indented once: left brace. Line 72, indented twice: labels, left square brace, row, right square brace, left square brace, c o l, right square brace, dot, set Background, left parenthesis, color, right parenthesis, semicolon. Line 73, indented once: right brace. Line 74: right brace.

Back to Example

Code for the Color Grid Game Controller class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Color Grid Game Controller class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x, dot, swing, dot, asterisk, semicolon. Line 6: import, java, dot, a w t, dot, asterisk, semicolon. Line 7: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 8: Blank. Line 9: public, class, Color Grid Game Controller, implements, Action Listener. Line 10: Left brace. Line 11, indented once: private, Color Grid Game, model, semicolon. Line 12, indented once: private, Bordered Label Grid View, view, semicolon. Line 13: Blank. Line 14, indented once: public, Color Grid Game Controller, left parenthesis, Color Grid Game, new Model, comma. Line 15, indented a few times: Bordered

Label Grid View, new View, right parenthesis. Line 16, indented once: left brace. Line 17, indented twice: model, equals, new Model, semicolon. Line 18, indented twice: view, equals, new View, semicolon. Line 19, indented twice: set Up Game, left parenthesis, right parenthesis, semicolon. Line 20, indented once: right brace. Line 21: Blank. Line 22, indented once: public, void, set Up Game, left parenthesis, right parenthesis. Line 23, indented once: left brace. Line 24, indented twice: view, dot, set Title, left parenthesis, model, dot, get Title, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 25, indented twice: set Up Labels, left parenthesis, right parenthesis, semicolon. Line 26, indented twice: set Up Buttons, left parenthesis, right parenthesis, semicolon. Line 27, indented once: right brace. Line 28: Blank. Line 29, indented once: public, void, set Up Labels, left parenthesis, right parenthesis. Line 30, indented once: left brace. Line 31, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, model, dot, get Size, left parenthesis, right parenthesis, semicolon, i, plus, plus, right parenthesis. Line 32, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, model, dot, get Size, left parenthesis, right parenthesis, semicolon, j, plus, plus, right parenthesis. Line 33, indented 4 times: view, dot, set Label Background, left parenthesis, i, comma, j, comma, model, dot, get Grid Color, left parenthesis, i, comma j, right parenthesis, right parenthesis, semicolon. Line 34, indented once: right brace. Line 35: Blank. Line 36, indented once: public, void, set Up Buttons, left parenthesis, right parenthesis. Line 37, indented once: left brace. Line 38, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, model, dot, get Number Of Colors, left parenthesis, right parenthesis, semicolon, i, plus, plus, right parenthesis. Line 39, indented twice: Left brace. Line 40, indented 3 times: view, dot, set Button Text, left parenthesis, i, comma, model, dot, get Label, left parenthesis, i, right parenthesis, right parenthesis, semicolon. Line 41, indented 3 times: view, dot, set Button Listener, left parenthesis, i, comma, this, right parenthesis, semicolon. Line 42, indented twice: right brace. Line 43, indented once: right brace. Line 44: Blank. Line 45, indented once: public, void, action Performed, left parenthesis, Action Event, a e, right parenthesis. Line 46, indented once: left brace. Line 47, indented twice: J Button, button, equals, left parenthesis, J Button, right parenthesis, a e, dot, get Source, left parenthesis, right parenthesis, semicolon. Line 48, indented twice: i n t, index, equals, model, dot, get Index, left parenthesis, button, dot, get Text, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 49: Blank. Line 50, indented twice: if, left parenthesis, model, dot, is Correct, left parenthesis, index, right parenthesis, right parenthesis. Line 51, indented 3 times: J Option Pane, dot, Message Dialog, left parenthesis, view, comma, open quotes, You won, close quotes, right parenthesis, semicolon. Line 52, indented twice: else. Line 53, indented 3 times: J Option Pane, dot, show Message

Dialog, left parenthesis, view, comma, open quotes, Sorry, comma, you lost, close quotes, right parenthesis, semicolon. Line 54, indented twice: Forward slash, forward slash, disable buttons here. Line 55, indented once: right brace. Line 56: right brace.

Back to Example

Code for the Play Color Count class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Play Color Count class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java x, dot, swing, dot, J Frame, semicolon. Line 6: import, java, dot, a w t, dot, Color, semicolon. Line 7: Blank. Line 8: public, class, Play Color Count. Line 9: Left brace. Line 10, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 11, indented once: Left brace. Line 12, indented twice: Color, left square brace, right square brace, colors, equals, left brace, Color, dot, RED, comma, Color, dot, GREEN, comma, Color, dot, BLUE, right brace, semicolon. Line 13, indented twice: String, left square brace, right square brace, labels, equals, left brace, open quotes, RED, close quotes, comma, open quotes, GREEN, close quotes, comma, open quotes, BLUE, close quotes, right brace, semicolon. Line 14, indented twice: Color Frequency Game, game. Line 15, indented 3 times: equals, new, Color Frequency Game, left parenthesis, 8, comma, colors, comma, labels, right parenthesis, semicolon. Line 16, indented twice: Bordered Label Grid View, game G u i. Line 17, indented 3 times: equals, new, Bordered Label Grid View, left parenthesis, colors, dot, length, comma, game, dot, get Size, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 18, indented twice: game G u i, dot, set Size, left parenthesis, 380, comma, 300, right parenthesis, semicolon Line 19, indented twice: game G u i, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 20, indented twice: Color Grid Game Controller, controller. Line 21, indented 3 times: equals, new, Color Grid Game Controller, left parenthesis, game, comma, game G u i, right parenthesis, semicolon. Line 22, indented once: right brace. Line 23: right brace.

Back to Example

A table detailing the Game class A P I. It has the following constructor. Class: Game. Constructor: Game, left parenthesis, i n t, n Sides, right parenthesis, instantiates a tile puzzle as an n Sides-by-n Sides grid. It has the following method. Return value: void. Method name and argument list. set Up Game, left parenthesis, i n t, n Sides, right parenthesis, resets the grid as an n Sides-by-n Sides grid.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, void, try, catch, static, String. Line 1: Forward slash. Asterisk. Java F X Shell Application. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot net dot U R L, semicolon. Line 6: import java f x dot application dot Application, semicolon. Line 7: import java f x dot f x m l dot F X M L Loader, semicolon. Line 8: import java f x dot scene dot layout dot H Box, semicolon. Line 9: import java f x dot scene dot Scene, semicolon. Line 10: import java f x dot stage dot Stage, semicolon. Line 11: blank. Line 12: public class F X Shell Application extends Application. Line 13: left brace. Line 14, indented once: at symbol, Override. Line 15, indented once: 2 forward slashes, start is main entry point for the application. Line 16, indented once: 2 forward slashes. It receives a Stage object, hyphen, the main window for the. Line 17, indented once: 2 forward slashes. G U I application. Line 18, indented once: public void start, left parentheses, Stage stage, right parentheses, 2 forward slashes, throws exception. Line 19, indented once: left brace. Line 20, indented twice: try. Line 21, indented twice: left brace. Line 22, indented thrice: 2 forward slashes. Locate the F X M L resource. Line 23, indented thrice: U R L u r l. Line 24, indented 4 times: equals, get Class, left parentheses, right parentheses, dot, get Resource, left parentheses, double quotes, f x m l underscore shell dot f x m l, double quotes, right parentheses, semicolon. Line 25: blank. Line 26, indented thrice: 2 forward slashes. Load the F X M L resource, instantiate root node, semicolon. Line 27, indented thrice: 2 forward slashes. use appropriate layout class for root node, semicolon. Line 28, indented thrice: 2 forward slashes, here we use H Box. Line 29, indented thrice: H Box root, equals, F X M L Loader dot load, left parentheses, u r l, right parentheses, semicolon. Line 30: blank. Line 31, indented thrice: 2 forward slashes, create a scene associated with the root. Line 32, indented thrice: 2 forward slashes, and set its width and height. Line 33, indented thrice: Scene scene, equals, new Scene, left parentheses, root, comma, 300, comma, 275, right parentheses, semicolon. Line 34: blank. Line 35, indented thrice: 2 forward slashes, assign the scene to the stage object. Line 36, indented thrice: stage dot set Scene, left parentheses, scene, right parentheses, semicolon. Line 37: blank. Line 38, indented thrice: 2 forward slashes, set title of stage, left parentheses, optional, right parentheses. Line 39, indented thrice: stage dot set Title, left parentheses, double quotes, Java F X Shell, double quotes, right parentheses, semicolon. Line 40: blank. Line 41, indented thrice: 2 forward slashes, make the stage visible. Line 42, indented thrice: stage dot show, left parentheses, right parentheses, semicolon. Line 43, indented twice: right brace. Line 44, indented twice: 2 forward slashes, The F X M L Loader load method throws an exception if. Line 45, indented twice: 2 forward slashes, the F X M L file is invalid or the U R L was not found. Line 46, indented twice: catch, left parentheses, Exception e, right parentheses. Line 47, indented twice: left brace. Line 48, indented thrice: e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 49, indented twice: right brace. Line 50, indented once: right brace. Line 51: blank. Line 52, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 53, indented once: left brace. Line 54, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 55, indented once: right brace. Line 56: right brace.

Back to Example

The methods are in the package java f x dot application. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: init, left parentheses, right parentheses, called by the Java F X runtime when the application is launched. Overriding this method is optional. Row 2. Return value: void. Method name and argument list: launch, left parentheses, String, dot, dot, dot, a r g s, right parentheses, static method that launches the application. Passes any command hyphen line arguments to the application. Row 3. Return value: Application dot Parameters. Method name and argument list: get Parameters, left parentheses, right parentheses, returns any command hyphen line arguments as an Application dot Parameters object. Row 4. Return value: void. Method name and argument list: start, left parentheses, Stage primary Stage, right parentheses, the main entry point for Java F X applications. The application places its scene onto the primary stage. This abstract method needs to be implemented. Row 5. Return value: void. Method name and argument list: stop, left parentheses, right parentheses, called when the application signals that it is ready to end. Overriding this method is optional.

Back to Table

The method is in the java dot lang package. The table has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Return value: U R L. Method name and argument list: get Resource, left parentheses, String resource, right parentheses, returns a U R L object representing the location of the resource or null if the resource is not found.

Back to Table

The method is in the java f x dot f x m l package. The table has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Return value: left angled bracket, uppercase t, right angled bracket, uppercase t. Method name and argument list: load, left parentheses, u r l location, right parentheses, static method that instantiates the nodes defined in the F X M L file specified by location and returns a reference to the root node.

Back to Table

The Scene method is in the java f x dot scene package. The constructor is as follows: Scene, left parentheses, Parent root, comma, double width, comma, double height, right parentheses, instantiates the Scene with a top level node of root and sets the width and height to the pixel values specified.

Back to Table

The methods are in the java f x dot stage package. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: set Scene, left parentheses, Scene scene, right parentheses, specifies the scene to be hosted by the Stage. Row 2. Return value: void. Method name and argument list: set Title, left parentheses, String title, right parentheses, sets the text to appear in the window title bar. Row 3. Return value: void. Method name and argument list: show, left parentheses, right parentheses, makes the window visible. By default, windows are not visible.

Back to Table

Line 1: left angled bracket, question mark, x m l version, equals, double quotes, 1.0, double quotes, encoding, equals, double quotes, U T F hyphen 8, double quotes, question mark, right angled bracket. Line 2: blank. Line 3.

left angled bracket, exclamation mark, 2 hyphens, import classes, 2 hyphens, right angled bracket. Line 4: left angled bracket, question mark, import java f x dot scene dot layout dot H Box, question mark, right angled bracket. Line 5: blank. Line 6: left angled bracket, exclamation mark, 2 hyphens, use appropriate layout manager, 2 hyphens, right angled bracket. Line 7: left angled bracket, H Box x m l n s, colon, f x, equals, double quotes, h t t p, colon, 2 forward slashes, java f x ot com, forward slash, f x m l, double quotes, right angled bracket. Line 8, indented once: left angled bracket, exclamation mark, 2 hyphens, define G U I components here, 2 hyphens, right angled bracket. Line 9: left angled bracket, forward slash, H Box, right angled bracket.

Back to Example

The G U I controls are in the java f x dot scene dot control package. The table has 2 columns labeled Java F X Class and Purpose. The row entries are as follows. Row 1. Java F X Class: Label. Purpose: Displays an image or read hyphen only text. Labels are often used to identify the contents of Text Fields. Row 2. Java F X Class: Text Field. Purpose: A single hyphen line text box for accepting user input. Row 3. Java F X Class: Button. Purpose: Command button that the user clicks to signal that an operation should be performed. Row 4. Java F X Class: Radio Button. Purpose: Toggle button that the user clicks to select one option in a group. Row 5. Java F X Class: Check Box. Purpose: Toggle button that the user clicks to select or deselect 0, comma, 1, comma, or more options in a group. Row 6. Java F X Class: Combo Box. Purpose: List of options from which the user selects one item. Row 7. Java F X Class: Slider. Purpose: Displays a set of continuous values along a horizontal or vertical line. The user can select a value by moving the knob or thumb.

Back to Table

The classes are in the java f x dot scene dot layout package. The table has 2 columns labeled Layout Container and Lays out its children nodes. The row entries are as follows. Row 1. Layout Container: H Box. Lays out its children nodes: In a single horizontal row. Row 2. Layout Container: V Box. Lays out its children nodes: In a single vertical column. Row 3. Layout Container: Border Pane. Lays out its children nodes: With at most one child in its top, left, right, bottom, and center positions. Row 4. Layout Container: Grid Pane. Lays out its children nodes: In a grid of rows and columns. A child can span more than one row or column. Row 5. Layout Container: Stack Pane. Lays out its children nodes: In a front to back stack.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, void, try, catch, new, static, String. Line 1: Forward slash. Asterisk. Displaying a Label and image. Line 2: Anderson, Franceschi. Line 3:

Asterisk. Forward slash. Line 4: blank. Line 5: import java dot net dot U R L, semicolon. Line 6: import java f x dot application dot Application, semicolon. Line 7: import java f x dot f x m l dot F X M L Loader, semicolon. Line 8: import java f x dot scene dot layout dot V Box, semicolon. Line 9: import java f x dot scene dot Scene, semicolon. Line 10: import java f x dot stage dot Stage, semicolon. Line 11: blank. Line 12: public class Dinner extends Application. Line 13: left brace. Line 14: blank. Line 15, indented once: at symbol, Override. Line 16, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 17, indented once: left brace. Line 18, indented twice: try. Line 19, indented twice: left brace. Line 20, indented thrice: 2 forward slashes, find the X M L resource. Line 21, indented thrice: U R L u r l. Line 22, indented 4 times: equals, get Class, left parentheses, right parentheses, dot, get Resource, left parentheses, double quotes, f x m l underscore dinner dot f x m l, double quotes, right parentheses, semicolon. Line 23, indented thrice: 2 forward slashes, load the X M L resource and instantiate the root node. Line 24, indented thrice: V Box root, equals, F X M L Loader dot load, left parentheses, u r l, right parentheses, semicolon. Line 25: blank. Line 26, indented thrice: 2 forward slashes, create a scene. Line 27, indented thrice: Scene scene, equals, new Scene, left parentheses, root, comma, 350, comma, 275, right parentheses, semicolon. Line 28: blank. Line 29, indented thrice: 2 forward slashes, set the scene. Line 30, indented thrice: stage dot set Scene, left parentheses, scene, right parentheses, semicolon. Line 31, indented thrice: 2 forward slashes, set title of stage. Line 32, indented thrice: stage dot set Title, left parentheses, double quotes, What's for dinner, question mark, double quotes, right parentheses, semicolon. Line 33, indented thrice: 2 forward slashes, show the stage. Line 34, indented thrice: stage dot show, left parentheses, right parentheses, semicolon. Line 35, indented twice: right brace. Line 36, indented twice: catch, left parentheses, Exception e, right parentheses. Line 37, indented twice: left brace. Line 38, indented thrice: e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 39, indented twice: right brace. Line 40, indented once: right brace. Line 41: blank. Line 42, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 43, indented once: left brace. Line 44, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 45, indented once: right brace. Line 46: right brace.

Back to Example

Line 1: left angled bracket, question mark, x m l version, equals, double quotes, 1.0, double quotes, encoding, equals, double quotes, U T F hyphen 8, double quotes, question mark, right angled bracket. Line 2: blank. Line 3. left angled bracket, question mark, import java f x dot scene dot control dot asterisk, question mark, right angled bracket. Line 4: left angled bracket, question mark, import java f x dot scene dot image dot asterisk, question mark, right angled bracket. Line 5: left angled bracket, question mark, import java f x dot scene dot layout dot asterisk, question mark, right angled bracket. Line 6: blank. Line 7: left angled bracket, V Box i d, equals, double quotes, root, double quotes, alignment, equals, double quotes, CENTER, double quotes, spacing, equals, double quotes, 25, double quotes, right angled bracket. Line 8, indented once: left angled bracket, Label text, equals, double quotes, Sushi tonight, question mark, double quotes, text Fill, equals, double quotes, BLUE, double quotes, forward slash, right angled bracket. Line 9, indented once: left angled bracket, Image View, right angled bracket. Line 10, indented twice: left angled bracket, Image u r l, equals, double quotes, at symbol, sushi dot j p g, double quotes, forward slash, right angled bracket. Line 11, indented once: left angled bracket, forward slash, Image View, right angled bracket. Line 12: left angled bracket, forward slash, V Box, right angled bracket.

Back to Example

Program code. The code contains the following keywords: public, class, static, double, return. Line 1: Forward slash. Asterisk. Simple Math Class. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Simple Math. Line 6: left brace. Line 7, indented once: Forward slash. 2 asterisks. square method. Line 8, indented once: asterisk. at symbol, param operand number to square. Line 9, indented once: asterisk, at symbol, return the square of operand. Line 10, indented once: Asterisk. Forward slash. Line 11, indented once: public static double square, left parentheses, double operand, right parentheses. Line 12, indented once: left brace. Line 13, indented twice: return operand asterisk operand, semicolon. Line 14, indented once: right brace. Line 15: blank. Line 16, indented once: Forward slash. 2 asterisks, cube method. Line 17, indented once: asterisk, at symbol, param operand number to cube. Line 18, indented once: asterisk. at symbol, return the cube of operand. Line 19, indented once: asterisk, forward slash. Line 20, indented once: public static double cube, left parentheses, double operand, right parentheses. Line 21, indented once: left brace. Line 22, indented twice: return operand asterisk operand asterisk operand, semicolon. Line 23, indented once: right brace. Line 24: right brace.

Back to Example

Line 1: left angled bracket, question mark, x m l, version, equals, double quotes, 1.0, double quotes, encoding, equals, double quotes, U T F hyphen 8, double quotes, question mark, right angled bracket. Line 2: blank. Line 3: left angled bracket, question mark, import java f x dot scene dot control dot asterisk, question mark, right angled bracket. Line 4: left angled bracket, question mark, import java f x dot scene dot layout dot asterisk, question mark, right angled bracket. Line 5: blank. Line 6: left angled bracket, V Box f x, colon, controller, equals, double quotes, Simple Math Controller, double quotes. Line 7, indented once: x m l n s, colon, f x, equals, double quotes, h t t p, colon, 2 forward slashes, java f x dot com, forward slash, f x m l, double quotes. Line 8, indented once: alignment, equals, double quotes, center, double quotes, spacing, equals, double quotes, 10, double quotes, right angled bracket. Line 9: blank. Line 10, indented once: left angled bracket, Label text, equals, double quotes, Enter a number, double quotes, forward slash, right angled bracket. Line 11, indented once: left angled bracket, Text Field f x, colon, i d, equals, double quotes, operand, double quotes, max Width, equals, double quotes, 100, double quotes, forward slash, right angled bracket. Line 12, indented once: left angled bracket, Button, f x, colon, i d, equals, double quotes, square, double quotes, text, equals, double quotes, Square, double quotes. Line 13, indented once: on Action, equals, double quotes, hash symbol, calculate, double quotes, forward slash, right angled bracket. Line 14, indented once: left angled bracket, Button f x, colon, i d, equals, double quotes, cube, double quotes, text, equals, double quotes, Cube, double quotes. Line 15, indented once: on Action, equals, double quotes, hash symbol, calculate, double quotes, forward slash, right angled bracket. Line 16, indented once: left angled bracket, Label text, equals, double quotes, Result, double quotes, forward slash, right angled bracket. Line 17, indented once: left angled bracket, Label f x, colon, i d, equals, double quotes, result, double quotes, forward slash, right angled bracket. Line 18: blank. Line 19: left angled bracket, forward slash, V Box, right angled bracket.

Back to Example

Program code. The code contains the following keywords: import, public, class, private, protected, void, try, catch, if, else. Line 1: import java f x dot event dot Action Event, semicolon. Line 2: import java f x dot f x m l dot F X M L, semicolon. Line 3: import java f x dot scene dot control dot asterisk, semicolon. Line 4: blank. Line 5: public class Simple Math Controller. Line 6: left brace. Line 7, indented once: at symbol, F X M L private Text Field operand, semicolon. Line 8, indented once: at symbol, F X M L, private Label result, semicolon. Line 9, indented once: at symbol, F X M L, private Button square, semicolon. Line 10, indented once: at symbol, F X M L private Button cube, semicolon. Line 11: blank. Line 12, indented once: at symbol, F X M L protected void calculate, left parentheses, Action Event event, right parentheses. Line 13, indented once: left brace. Line 14, indented twice: try. Line 15, indented twice: left brace. Line 16, indented thrice: double o p, equals, Double dot parse Double, left parentheses, operand dot get Text, left parentheses, right parentheses, right parentheses, semicolon. Line 17, indented twice: if, left parentheses, event dot get Source, left parentheses, right parentheses, 2 equals sign, square, right parentheses. Line 18, indented thrice: result dot set Text, left parentheses, String dot value Of, left parentheses, Simple Math dot square, left parentheses, o p, right parentheses, right parentheses, right parentheses, semicolon. Line 19, indented twice: else if, left parentheses, event dot get Source, left parentheses, right parentheses, equals, equals, cube, right parentheses. Line 20, indented thrice: result dot set Text, left parentheses, String dot value Of, left parentheses, Simple Math dot cube, left parentheses, o p, right parentheses, right parentheses, right parentheses, semicolon. Line 21, indented twice: right brace. Line 22, indented twice: catch, left parentheses, Number Format Exception n f e, right parentheses. Line 23, indented twice: left brace. Line 24, indented thrice: operand dot set Text, left parentheses, double quotes, double quotes, right parentheses, semicolon. Line 25, indented thrice: result dot set Text, left parentheses, double quotes, 3 question marks, double quotes, right parentheses, semicolon. Line 26, indented twice: right brace. Line 27, indented once: right brace. Line 28: right brace.

Back to Example

The methods are in the java f x dot scene dot control package. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: String. Method name and argument list: get Text, left parentheses, right parentheses, returns the text typed into the Text Field. Row 2. Return value: void. Method name and argument list: set Text, left parentheses, String new Text, right parentheses, sets the text in the Text Field to new Text.

Back to Table

The method is in the java f x dot event package. The method has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Return value: Object. Method name and argument list: get Source, left parentheses, right parentheses, returns the object on which the event was triggered.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, public, void, try, catch, static, String. Line 1: Forward slash. Asterisk. Simple Math Operations Using Buttons. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot net dot U R L, semicolon. Line 6: import java f x dot application dot Application, semicolon. Line 7: import java f x dot f x m l dot F X M L Loader, semicolon. Line 8: import java f x dot scene dot layout dot V Box, semicolon. Line 9: import java f x dot scene dot Scene, semicolon. Line 10: import java f x dot stage dot Stage, semicolon. Line 11: blank. Line 12: public class Simple Math Practice extends Application. Line 13: left brace. Line 14, indented once: at symbol, Override. Line 15, indented once: public void start, left parentheses, Stage stage, right parentheses, 2 forward slashes, throws Exception. Line 16, indented once: left brace. Line 17, indented twice: try. Line 18, indented twice: left brace. Line 19, indented thrice: U R L u r l, equals. Line 20, indented 4 times: get Class, left parentheses, right parentheses, dot, get Resource, left parentheses, double quotes, f x m l, underscore, simple, underscore, math, dot, f x m l, double quotes, right parentheses, semicolon. Line 21, indented thrice: V Box root, equals, F X M L Loader dot load, left parentheses, u r l, right parentheses, semicolon. Line 22, indented thrice: Scene scene, equals, new Scene, left parentheses, root, comma, 300, comma, 275, right parentheses, semicolon. Line 23, indented thrice: stage dot set Title, left parentheses, double quotes, Simple Math, double quotes, right parentheses, semicolon. Line 24, indented thrice: stage dot set Scene, left parentheses, scene, right parentheses, semicolon. Line 25, indented thrice: stage dot show, left parentheses, right parentheses, semicolon. Line 26, indented twice: right brace. Line 27, indented twice: catch, left parentheses, Exception e, right parentheses. Line 28, indented twice: left brace. Line 29, indented 4 times: e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 30, indented twice: right brace. Line 31, indented once: right brace. Line 32: blank. Line 33, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 34, indented once: left brace. Line 35, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 36, indented once: right brace. Line 37: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, final, int, String, if, else, return. Line 1: Forward slash. Asterisk. Color Selector class. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Color Selector. Line 6: left brace. Line 7, indented once: public static final int RED, equals, 0, semicolon. Line 8, indented once: public static final int GREEN, equals, 1, semicolon. Line 9, indented once: public static final int BLUE, equals, 2, semicolon. Line 10: blank. Line 11, indented once: Forward slash. 2 asterisks. color To Hex String method. Line 12, indented once: asterisk. at symbol, param, selection the selected color. Line 13, indented once: asterisk. at symbol, return the hex representation of the selected color. Line 14, indented once: asterisk. Forward slash. Line 15, indented once: public static String color To Hex String, left parentheses, int selection, right parentheses. Line 16, indented once: left brace. Line 17, indented twice: String result, equals, double quotes, hash symbol, double quotes, semicolon. Line 18, indented twice: if, left parentheses, selection, 2 equals sign, RED, right parentheses. Line 19, indented thrice: result plus, equals, double quotes, F F 0 0 0 0, double quotes, semicolon. Line 20, indented twice: else if, left parentheses, selection, 2 equals sign, GREEN, right parentheses. Line 21, indented thrice: result, plus, equals, double quotes, 0 0 F F 0 0, double quotes, semicolon. Line 22, indented twice: else if, left parentheses, selection, 2 equals sign, BLUE, right parentheses. Line

23, indented thrice: result, plus, equals, double quotes, 0 0 0 0 F F, double quotes, semicolon. Line 24, indented twice: return result, semicolon. Line 25, indented once: right brace. Line 26: right brace.

Back to Example

Line 1: left angled bracket, question mark, x m l version, equals, double quotes, 1.0, double quotes, encoding, equals, double quotes, U T F hyphen 8, double quotes, question mark, right angled bracket. Line 2: blank. Line 3: left angled bracket, question mark, import java f x dot scene dot control dot asterisk, question mark, right angled bracket. Line 4: left angled bracket, question mark, import java f x dot scene dot layout dot asterisk, question mark, right angled bracket. Line 5: blank. Line 6: left angled bracket, H Box f x, colon, controller, equals, double quotes, Changing Colors Controller, double quotes. Line 7, indented twice: x m l n s, colon, f x, equals, double quotes, h t t p, colon, 2 forward slashes, java f x dot com, forward slash, f x m l, double quotes. Line 8, indented twice: alignment, equals, double quotes, center, double quotes, spacing, equals, double quotes, 10, double quotes, right angled bracket. Line 9: blank. Line 10, indented once: left angled bracket, f x, colon, define, right angled bracket. Line 11, indented twice: left angled bracket, Toggle Group f x, colon, i d, equals, double quotes, color Group, double quotes, forward slash, right angled bracket. Line 12, indented once: left angled bracket, forward slash, f x, colon, define, right angled bracket. Line 13: blank. Line 14, indented once: left angled bracket, Radio Button f x, colon, i d, equals, double quotes, red, double quotes, text, equals, double quotes, red, double quotes, selected, equals, double quotes, true, double quotes. Line 15, indented multiple times: toggle Group, equals, double quotes, dollar sign, color Group, double quotes. Line 16, indented multiple times: on Action, equals, double quotes, hash symbol, color Chosen, double quotes, forward slash, right angled bracket. Line 17, indented once: left angled bracket, Radio Button f x, colon, i d, equals, double quotes, green, double quotes, text, equals, double quotes, green, double quotes. Line 18, indented multiple times: toggle Group, equals, double quotes, dollar sign, color Group, double quotes. Line 19, indented multiple times: on Action, equals, double quotes, hash symbol, color Chosen, double quotes, forward slash, right angled bracket. Line 20, indented once: left angled bracket, Radio Button f x, colon, i d, equals, double quotes, blue, double quotes, text, equals, double quotes, blue, double quotes. Line 21, indented multiple times: toggle Group, equals, double quotes, dollar sign, color Group, double quotes. Line 22, indented multiple times: on Action, equals, double quotes, hash symbol, color Chosen, double quotes, forward slash, right angled bracket. Line 23: blank. Line 24, indented once: left angled bracket, Label f x, colon, i d, equals, double quotes, label, double quotes, text, equals, double quotes, Watch my background, double quotes. Line 25, indented multiple times: text Fill, equals, double quotes, WHITE, double quotes. Line 26, indented multiple times: style, equals, double quotes, hyphen, f x, hyphen, background, hyphen, color, colon, hash symbol, F F 0 0 0 0, double quotes, forward slash, right angled bracket. Line 27: blank. Line 28: left angled bracket, forward slash, H Box, right angled bracket.

Back to Example

The table summarizes the meaning of the element, its attributes, the annotation and the prefixes of F X M L. The row entries are as follows. Row 1. F X M L Element: f x colon define. Meaning: Used to create objects not in the scene graph that need to be referenced later. Row 2. F X M L Attributes: f x colon i d. Meaning: Defines a name that can be referenced across the F X M L application. Row 3. F X M L Attributes: f x colon controller. Meaning: Defines a class that contains event handling code for one or more controls. Row 4. F X M L Attributes: on Action. Meaning: Defines a method name, left parentheses, preceded by, double quotes, hash symbol, double quotes, right parentheses, in the controller that should be executed when the user interacts with the control. Row 5. F X M L Annotation: at symbol, F X M L. Meaning: Allows F X M L to access a private or protected class, method, or data. Row 6. F X M L Prefixes: hyphen f x hyphen. Meaning: Used to distinguish a Java F X style attribute from a C S S attribute. Row 7. F X M L Prefixes: dollar sign. Meaning: Used as a prefix to a variable name when the variable is used as a property value. Row 8. F X M L Prefixes: at symbol. Meaning: Used as a prefix for a U R I to specify that the path of the U R I starts with the current folder.

Back to Table

Program code. The code contains the following keywords: import, public, class, private, protected, void, String, if, else. Line 1: Forward slash. Asterisk. Changing Colors Controller class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java f x dot event dot Action Event semicolon. Line 6: import java f x dot f x m l dot F X M L semicolon. Line 7: import java f x dot scene dot control dot Asterisk semicolon. Line 8: blank. Line 9: public class Changing Colors Controller. Line 10: Left brace. Line 11: at symbol, F X M L private Radio Button red semicolon. Line 12: at symbol, F X M L private Radio Button green semicolon. Line 13: at symbol, F X M L private Radio Button blue semicolon. Line 14: at symbol, F X M L private Label label semicolon. Line 15: blank. Line 16: at symbol, F X M L protected void color Chosen left parentheses Action Event event right parentheses. Line 17: Left brace. Line 18: String style equals double quotes hyphen f x hyphen background hyphen color colon double quotes semicolon. Line 19: blank. Line 20: if left parentheses event dot get Source left parentheses right parentheses equals equals red right parentheses. Line 21: style plus equals. Line 22: Color Selector dot color To Hex String left parentheses Color Selector dot RED right parentheses semicolon. Line 23: else if left parentheses event dot get Source left parentheses right parentheses equals equals green right parentheses. Line 24: style plus equals. Line 25: Color Selector dot color To Hex String left parentheses Color Selector dot GREEN right parentheses semicolon. Line 26: else if left parentheses event dot get Source left parentheses right parentheses equals equals blue right parentheses. Line 27: style plus equals. Line 28: Color Selector dot color To Hex String left parentheses Color Selector dot BLUE right parentheses semicolon. Line 29: blank. Line 30: label dot set Style left parentheses style right parentheses semicolon. Line 31: Right brace. Line 32: Right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, void, try, new, catch, static, String. Line 1: Forward slash. Asterisk. Select a Color Using Radio Buttons. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot net dot U R L semicolon. Line 6: import java f x dot application dot Application semicolon. Line 7: import java f x dot f x m l dot F X M L Loader semicolon. Line 8: import java f x dot scene dot layout dot H Box semicolon. Line 9: import java f x dot scene dot Scene semicolon. Line 10: import java f x dot stage dot Stage semicolon. Line 11: blank. Line 12: public class Changing Colors extends Application. Line 13: Left brace. Line 14: at symbol, Override. Line 15: public void start left parentheses Stage stage right parentheses. Line 16: Left brace. Line 17: try. Line 18: Left brace. Line 19: URL u r l equals. Line 20: get Class left parentheses right parentheses dot get Resource left parentheses double quotes f x m l underscore changing underscore colors dot f x m l double quotes right parentheses semicolon. Line 21: H Box root equals F X M L Loader dot load left parentheses u r l right parentheses semicolon. Line 22: Scene scene equals new Scene left parentheses root comma 400 comma 200 right parentheses semicolon. Line 23: stage dot set Title left parentheses double quotes Selecting a color double quotes right parentheses semicolon. Line 24: stage dot set Scene left parentheses scene right parentheses semicolon. Line 25: stage dot show left parentheses right parentheses semicolon. Line 26: Right brace. Line 27: catch left parentheses Exception e right parentheses. Line 28: Left brace. Line 29: e dot print Stack Trace left parentheses right parentheses semicolon. Line 30: Right brace. Line 31: Right brace. Line 32: blank. Line 33: public static void main left parentheses String left square bracket right square bracket a r g s right parentheses. Line 34: Left brace. Line 35: launch left parentheses a r g s right parentheses semicolon. Line 36: Right brace. Line 37: Right brace.

Back to Example

Program code. The code contains the following keywords: public, class, static, final, int, boolean, private, true, false, return. Line 1: Forward slash. Asterisk. Color Mixer class. Line 2: Anderson comma Franceschi. Line 3: Asterisk.

Forward slash. Line 4: blank. Line 5: public class Color Mixer. Line 6: Left brace. Line 7: public static final int RED equals 0 semicolon. Line 8: public static final int GREEN equals 1 semicolon. Line 9: public static final int BLUE equals 2 semicolon. Line 10: blank. Line 11: private boolean left square bracket right square bracket r g b semicolon. Line 12: blank. Line 13: Forward slash. Asterisk. Asterisk default constructor. Line 14: Asterisk sets all elements in the r g b array to true. Line 15: Asterisk to represent the color white. Line 16: Asterisk. Forward slash. Line 17: public Color Mixer left parentheses right parentheses. Line 18: Left brace. Line 19: r g b equals new Boolean left square bracket 3 right square bracket semicolon. Line 20: r g b left square bracket RED right square bracket equals true semicolon. Line 21: r g b left square bracket GREEN right square bracket equals true semicolon. Line 22: r g b left square bracket BLUE right square bracket equals true semicolon. Line 23: Right brace. Line 24: blank. Line 25: Forward slash. Asterisk. Asterisk toggle Color method. Line 26: Asterisk toggles the color on Forward slash off. Line 27: Asterisk at symbol, param color the color to be toggled. Line 28: Asterisk. Forward slash. Line 29: public void toggle Color left parentheses int color right parentheses. Line 30: Left brace. Line 31: r g b left square bracket color right square bracket equals exclamation mark r g b left square bracket color right square bracket semicolon. Line 32: Right brace. Line 33: blank. Line 34: Forward slash. Asterisk. Asterisk hex String Color. Line 35: Asterisk at symbol, return the hexadecimal representation. Line 36: Asterisk of the color mix. Line 37: Asterisk. Forward slash. Line 38: public String hex String Color left parentheses right parentheses. Line 39: Left brace. Line 40: String result equals double quotes hash double quotes semicolon. Line 41: blank. Line 42: result plus equals left parentheses r g b left square bracket RED right square bracket question mark double quotes F F double quotes colon double quotes 0 0 double quotes right parentheses semicolon. Line 43: result plus equals left parentheses r g b left square bracket GREEN right square bracket question mark double quotes F F double quotes colon double quotes 0 0 double quotes right parentheses semicolon. Line 44: result plus equals left parentheses r g b left square bracket BLUE right square bracket question mark double quotes F F double quotes colon double quotes 0 0 double quotes right parentheses semicolon. Line 45: blank. Line 46: return result semicolon. Line 47: Right brace. Line 48: Right brace.

Back to Example

Line 1: Left angled bracket question mark x m l version equals double quotes 1 dot 0 double quotes encoding equals double quotes U T F hyphen 8 double quotes question mark Right angled bracket. Line 2: blank. Line 3: Left angled bracket question mark import java f x dot scene dot control dot Asterisk question mark Right angled bracket. Line 4: Left angled bracket question mark import java f x dot scene dot layout dot Asterisk question mark

Right angled bracket. Line 5: blank. Line 6: Left angled bracket H Box f x colon controller equals double quotes Mixing Colors Controller double quotes. Line 7: x m l n s colon f x equals double quotes http colon Forward slash Forward slash java f x dot com Forward slash f x m l double quotes. Line 8: alignment equals double quotes center double quotes spacing equals double quotes 10 double quotes. Line 9: style equals double quotes hyphen f x hyphen background hyphen color colon hash C C C C C C double quotes Right angled bracket. Line 10: blank. Line 11: Left angled bracket Check Box f x colon id equals double quotes red double quotes selected equals double quotes true double quotes. Line 12: text equals double quotes red double quotes on Action equals double quotes hash mix double quotes Forward slash Right angled bracket. Line 13: Left angled bracket Check Box f x colon i d equals double quotes green double quotes selected equals double quotes true double quotes. Line 14: text equals double quotes green double quotes on Action equals double quotes hash mix double quotes Forward slash Right angled bracket. Line 15: Left angled bracket Check Box f x colon i d equals double quotes blue double quotes selected equals double quotes true double quotes. Line 16: text equals double quotes blue double quotes on Action equals double quotes hash mix double quotes Forward slash Right angled bracket. Line 17: blank. Line 18: Left angled bracket Label f x colon id equals double quotes label double quotes text equals double quotes Watch my background double quotes. Line 19: style equals double quotes hyphen f x hyphen background hyphen color colon hash F F F F F F double quotes Forward slash Right angled bracket. Line 20: blank. Line 21: Left angled bracket Forward slash H Box Right angled bracket.

Back to Example

Program code. The code contains the follwing keywords: import, public, class, private, new, if, else, String. Line 1: Forward slash. Asterisk. Mixing Colors Controller class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java f x dot event dot Action Event semicolon. Line 6: import java f x dot f x m l dot F X M L semicolon. Line 7: import java f x dot scene dot control dot Asterisk semicolon. Line 8: blank. Line 9: public class Mixing Colors Controller. Line 10: Left brace. Line 11: private Color Mixer mixer semicolon. Line 12: blank. Line 13: at symbol, F X M L private Check Box red semicolon. Line 14: at symbol, F X M L private Check Box green semicolon. Line 15: at symbol, F X M L private Check Box blue semicolon. Line 16: at symbol, F X M L private Label label semicolon. Line 17: blank. Line 18: public Mixing Colors Controller left parentheses right parentheses. Line 19: Left brace. Line 20: mixer equals new Color Mixer left parentheses right parentheses semicolon. Line 21: Right brace. Line 22: blank. Line 23: at symbol, F X M L protected void mix left parentheses Action Event event right parentheses. Line 24: Left brace. Line 25: if left parentheses event dot get

Source left parentheses right parentheses equals equals red right parentheses. Line 26: mixer dot toggle Color left parentheses Color Mixer dot RED right parentheses semicolon. Line 27: else if left parentheses event dot get Source left parentheses right parentheses equals equals green right parentheses. Line 28: mixer dot toggle Color left parentheses Color Mixer dot GREEN right parentheses semicolon. Line 29: else if left parentheses event dot get Source left parentheses right parentheses equals equals blue right parentheses. Line 30: mixer dot toggle Color left parentheses Color Mixer dot BLUE right parentheses semicolon. Line 31: blank. Line 32: String style equals double quotes hyphen f x hyphen background hyphen color colon double quotes semicolon. Line 33: style plus equals mixer dot hex String Color left parentheses right parentheses semicolon. Line 34: label dot set Style left parentheses style right parentheses semicolon. Line 35: Right brace. Line 36: Right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, try, void, catch, new, static. Line 1: Forward slash. Asterisk. Mixing Colors Using Check Boxes. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot net dot U R L semicolon. Line 6: import java f x dot application dot Application semicolon. Line 7: import java f x dot f x m l dot F X M L Loader semicolon. Line 8: import java f x dot scene dot layout dot H Box semicolon.

Line 9: import java f x dot scene dot Scene semicolon. Line 10: import java f x dot stage dot Stage semicolon. Line 11: blank. Line 12: public class Mixing Colors extends Application. Line 13: Left brace. Line 14: at symbol, Override. Line 15: public void start left parentheses Stage stage right parentheses. Line 16: Left brace. Line 17: try. Line 18: Left brace. Line 19: U R L u r l equals. Line 20: get Class left parentheses right parentheses dot get Resource left parentheses double quotes f x m l underscore mixing underscore colors dot f x m l double quotes right parentheses semicolon. Line 21: H Box root equals F X M L Loader dot load left parentheses u r l right parentheses semicolon. Line 22: Scene scene equals new Scene left parentheses root comma 400 comma 200 right parentheses semicolon. Line 23: stage dot set Title left parentheses double quotes Mixing colors double quotes right parentheses semicolon. Line 24: stage dot set Scene left parentheses scene right parentheses semicolon. Line 25: stage dot show left parentheses right parentheses semicolon. Line 26: Right brace. Line 27: catch left parentheses Exception e right parentheses. Line 28: Left brace. Line 29: e dot print Stack Trace left parentheses right parentheses semicolon. Line 30: Right brace. Line 31: Right brace. Line 32: blank. Line 33: public static void main left parentheses String left square bracket right square bracket a r g s right parentheses. Line 34: Left brace. Line 35: launch left parentheses a r g s right parentheses semicolon. Line 36: Right brace. Line 37: Right brace.

Back to Example

Line 1: Left angled bracket Question mark x m l version equals double quotes 1.0 double quotes encoding equals double quotes U T F hyphen 8 double quotes Question mark Right angled bracket. Line 2: blank. Line 3: Left angled bracket Question mark import java f x dot scene dot asterisk Question mark Right angled bracket. Line 4: Left angled bracket Question mark import java f x dot scene dot control. asterisk Question mark Right angled bracket. Line 5: Left angled bracket Question mark import java f x dot scene dot layout dot asterisk Question mark Right angled bracket. Line 6: blank. Line 7: Left angled bracket exclamation mark hyphen hyphen asterisk asterisk asterisk asterisk asterisk 1. Student code starts here hyphen hyphen Right angled bracket. Line 8: Left angled bracket exclamation mark hyphen hyphen add the code to this element definition to specify. Line 9: the controller as Button Practice Controller hyphen hyphen Right angled bracket. Line 10: Left angled bracket Border Pane f x colon i d equals double quotes b p double quotes. Line 11: x m l n s colon f x equals double quotes h t t p colon Forward slash Forward slash java f x dot com Forward slash f x m l double quotes Right angled bracket. Line 12: blank. Line 13: Left angled bracket top Right angled bracket. Line 14: Left angled bracket exclamation mark hyphen hyphen asterisk asterisk asterisk asterisk asterisk 2. Student code restarts here hyphen hyphen Right angled bracket. Line 15: Left angled bracket exclamation mark hyphen hyphen add code to define an H Box layout that is centered. Line 16: with an i d of h Box and 10 pixels between controls. Line 17: Inside the H Box, add 2 Buttons, with text OPEN and. Line 18: CLOSE, and i d s of open and close. Clicking on either. Line 19: button triggers a call to the event handler flip. Line 20: method hyphen hyphen Right angled bracket. Line 21: blank. Line 22: blank. Line 23: blank. Line 24: Left angled bracket Forward slash top Right angled bracket. Line 25: blank. Line 26: Left angled bracket Forward slash Border Pane Right angled bracket.

Back to Example

Line 48: Forward slashForward slash asterisk asterisk asterisk asterisk asterisk 3. Student code restarts here. Line 49: Forward slash Forward slash Code the flip method. Line 50: Forward slash Forward slash To open the switch, call the open method. Line 51: Forward slash Forward slash with circuit, the object reference of the. Line 52: Forward slash Forward slash Circuit object. Line 53: Forward slash Forward slash The open method does not take any parameters. Line 54: Forward slash Forward slash To close the switch, call the close method. Line 55: Forward slash Forward slash with circuit. Line 56: Forward slash Forward slash The close method does not take any parameters. Line 57: Forward slash Forward slash The last statement of the method should be. Line 58: Forward slash Forward slash animate left parentheses, right parentheses, semicolon.

Back to Example

Program code. The code contains the following keywords: import, public, class, private, String, int, return, void. Line 1: Forward slash. Asterisk. Food Sampler class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java f x dot scene dot image dot Image semicolon. Line 6: blank. Line 7: public class Food Sampler. Line 8: left brace. Line 9: private String left square bracket, right square bracket, country List equals. Line 10: left brace double quotes France double quotes comma double quotes Greece double quotes comma double quotes Italy double quotes comma double quotes Japan double quotes comma double quotes USA double quotes right brace semicolon. Line 11: private Image left square bracket, right square bracket foods equals. Line 12: left brace new Image left parentheses double quotes cheese dot j p g double quotes right parentheses comma. Line 13: new Image left parentheses double quotes f e t a salad dot j p g double quotes right parentheses comma. Line 14: new Image left parentheses double quotes pizza dot j p g double quotes right parentheses comma. Line 15: new Image left parentheses double quotes sushi dot j p g double quotes right parentheses comma. Line 16: new Image left parentheses double quotes hamburger dot j p g double quotes right parentheses right brace semicolon. Line 17: private int selected Index equals 0 semicolon. Line 18: blank. Line 19: Forward slash. Asterisk. Asterisk get

Country List method. Line 20: Asterisk at symbol return a reference to the country List array. Line 21: Asterisk. Forward slash. Line 22: public String left square bracket, right square bracket get Country List left parentheses right parentheses. Line 23: left brace. Line 24: return country List semicolon. Line 25: right brace. Line 26: blank. Line 27: Forward slash. Asterisk. Asterisk the accessor for selected Index. Line 28: Asterisk at symbol return the index selected. Line 29: Asterisk. Forward slash. Line 30: public int get Selected Index left parentheses right parentheses. Line 31: left brace. Line 32: return selected Index semicolon. Line 33: right brace. Line 34: blank. Line 35: Forward slash. Asterisk. Asterisk the mutator for selected Index. Line 36: Asterisk at symbol param selection the new value for selected Index. Line 37: Asterisk. Forward slash. Line 38: public void update Selection left parentheses int selection right parentheses. Line 39: left brace. Line 40: selected Index equals selection semicolon. Line 41: right brace. Line 42: blank. Line 43: Forward slash. Asterisk. Asterisk the accessor for the image to display. Line 44: Asterisk at symbol return the image from the Image array. Line 45: Asterisk using the index selected Index. Line 46: Asterisk. Forward slash. Line 47: public Image get Image Selected left parentheses right parentheses. Line 48: left brace. Line 49: return foods left square bracket selected Index right square bracket semicolon. Line 50: right brace. Line 51: right brace.

Back to Example

Line 1: Left angled bracket Question mark x m l version equals double quotes 1 dot 0 double quotes encoding equals double quotes U T F hyphen 8 double quotes Question mark Right angled bracket. Line 2: blank. Line 3: Left angled bracket Question mark import java f x dot scene dot control dot Asterisk Question mark Right angled bracket. Line 4: Left angled bracket Question mark import java f x dot scene dot image dot Asterisk Question mark Right angled bracket. Line 5: Left angled bracket Question mark import java f x dot scene dot layout dot Asterisk Question mark Right angled bracket. Line 6: blank. Line 7: Left angled bracket H Box f x colon controller equals double quotes Food Samplings Controller double quotes. Line 8: x m l n s colon f x equals double quotes h t t p colon Forward slash Forward slash java f x dot com Forward slash f x m l double quotes. Line 9: alignment equals double quotes center double quotes spacing equals double quotes 10 double quotes Right angled bracket. Line 10: blank. Line 11: Left angled bracket Combo Box f x colon i d equals double quotes countries double quotes visible Row Count equals double quotes 3 double quotes. Line 12: on Action equals double quotes hash item Selected double quotes Right angled bracket. Line 13: Left angled bracket Forward slash Combo Box Right angled bracket. Line 14: blank. Line 15: Left angled bracket Image View f x colon i d equals double quotes food Image double quotes Right angled bracket. Line 16: Left angled bracket Forward slash Image View Right angled bracket. Line 17: blank. Line 18: Left angled bracket Forward slash H Box Right angled bracket.

Back to Example

Program code. The code contains the following keywords: import, public, class, private, void, new, protected, int. Line 1: Forward slash. Asterisk. Food Samplings Controller class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java f x dot event dot Action Event semicolon. Line 6: import java f x dot f x m l dot F X M L semicolon. Line 7: import java f x dot scene dot control dot Asterisk semicolon. Line 8: import java f x dot scene dot image dot Asterisk semicolon. Line 9: blank. Line 10: public class Food Samplings Controller. Line 11: left brace. Line 12: private Food Sampler sampler semicolon. Line 13: blank. Line 14: at symbol F X M L private Combo Box Left angled bracket String Right angled bracket countries semicolon. Line 15: at symbol F X M L private Image View food Image semicolon. Line 16: private Single Selection Model Left angled bracket String Right angled bracket selection Model semicolon. Line 17: blank. Line 18: public void initialize left parentheses right parentheses. Line 19: left brace. Line 20: sampler equals new Food Sampler left parentheses right parentheses semicolon. Line 21: blank. Line 22: Forward slash Forward slash populate combo box with data from the Model. Line 23: countries dot get

Items left parentheses right parentheses dot add All left parentheses sampler dot get Country List left parentheses right parentheses right parentheses semicolon. Line 24: blank. Line 25: Forward slash Forward slash get a reference to the Single Selection Model. Line 26: selection Model equals countries dot get Selection Model left parentheses right parentheses semicolon. Line 27: blank. Line 28: Forward slash Forward slash initialize View with initial data from Model. Line 29: selection Model dot select left parentheses sampler dot get Selected Index left parentheses right parentheses right parentheses semicolon. Line 30: food Image dot set Image left parentheses sampler dot get Image Selected left parentheses right parentheses right parentheses semicolon. Line 31: right brace. Line 32: blank. Line 33: at symbol F X M L protected void item Selected left parentheses Action Event event right parentheses. Line 34: left brace. Line 35: Forward slash Forward slash retrieve index of country selected. Line 36: int index equals selection Model dot get Selected Index left parentheses right parentheses semicolon. Line 37: Forward slash Forward slash update the Model. Line 38: sampler dot update Selection left parentheses index right parentheses semicolon. Line 39: Forward slash Forward slash update the View with Image from the Model. Line 40: food Image dot set Image left parentheses sampler dot get Image Selected left parentheses right parentheses right parentheses semicolon. Line 41: right brace. Line 42: right brace.

Back to Example

The methods are in the java f x dot scene dot control package. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: int. Method name and argument list: get Selected Index, left parentheses, right parentheses, returns the index of the currently selected item. Row 2. Return value: uppercase t. Method name and argument list: get Selected Item, left parentheses, right parentheses, returns the currently selected item. Row 3. Return value: void. Method name and argument list: select, left parentheses, int index, right parentheses, selects the item at index. Any previously selected item is deselected.

Back to Table

The method is in the java f x dot scene dot control package. The table has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Return value: Observable List. Method name and argument list: get Items, left parentheses, right parentheses, returns the values of the items in the Combo Box as an Observable List.

Back to Table

The method is in the java f x dot collections package. The table has 2 columns labeled Return value and Method name and argument list. The row entry is as follows.

Return value: boolean. Method name and argument list: add All, left parentheses, uppercase e, dot, dot, dot, elements, right parentheses, adds the elements to the list.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, void, try, new, catch. Line 1: Forward slash. Asterisk. Using Combo Box to show a sampling of international foods. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot net dot U R L semicolon. Line 6: import java f x dot application dot Application semicolon. Line 7: import java f x dot f x m l dot F X M L Loader semicolon. Line 8: import java f x dot scene dot layout dot H Box semicolon. Line 9: import java f x dot scene dot Scene semicolon. Line 10: import java f x dot stage dot Stage semicolon. Line 11: blank. Line 12: public class Food Samplings extends Application. Line 13: left brace. Line 14: at symbol Override. Line 15: public void start left parentheses Stage stage right parentheses. Line 16: left brace. Line 17: try. Line 18: left brace. Line 19: U R L u r l equals. Line 20: get Class left parentheses right parentheses dot get Resource left parentheses double quotes f x m l underscore food underscore samplings dot f x m l double quotes right parentheses semicolon. Line 21: H Box root equals F X M L Loader dot load left parentheses u r l right parentheses semicolon. Line 22: Scene scene equals new Scene left parentheses root comma 450 comma 200 right parentheses semicolon.

Line 23: stage dot set Title left parentheses double quotes Food samplings of various countries double quotes right parentheses semicolon. Line 24: stage dot set Scene left parentheses scene right parentheses semicolon. Line 25: stage dot show left parentheses right parentheses semicolon. Line 26: right brace. Line 27: catch left parentheses Exception e right parentheses. Line 28: left brace. Line 29: e dot print Stack Trace left parentheses right parentheses semicolon. Line 30: right brace. Line 31: right brace. Line 32: blank. Line 33: public static void main left parentheses String left square bracket, right square bracket a r g s right parentheses. Line 34: left brace. Line 35: launch left parentheses a r g s right parentheses semicolon. Line 36: right brace. Line 37: right brace.

Back to Example

Line 1: Left angled bracket Question mark x m l version equals double quotes 1 dot 0 double quotes encoding equals double quotes U T F hyphen 8 double quotes Question mark Right angled bracket. Line 2: blank. Line 3: Left angled bracket Question mark import java f x dot scene dot control dot Asterisk Question mark Right angled bracket. Line 4: Left angled bracket Question mark import java f x dot scene dot image dot Asterisk Question mark Right angled bracket. Line 5: Left angled bracket Question mark import java f x dot scene dot layout dot Asterisk Question mark Right angled bracket. Line 6: blank. Line 7: Left angled bracket V Box f x colon controller equals double quotes Photo Grayer Controller double quotes. Line 8: x m l n s colon f x equals double quotes h t t p colon Forward slash Forward slash java f x dot com Forward slash f x m l double quotes. Line 9: alignment equals double quotes center double quotes spacing equals double quotes 10 double quotes. Line 10: style equals double quotes hyphen f x hyphen padding colon 0 10 0 10 semicolon double quotes Right angled bracket. Line 11: blank. Line 12: Left angled bracket Image View f x colon i d equals double quotes original Image View double quotes Right angled bracket. Line 13: Left angled bracket Image u r l equals double quotes Sushi dot j p g double quotes Forward slash Right angled bracket. Line 14: Left angled bracket Forward slash Image View Right angled bracket. Line 15: blank. Line 16: Left angled bracket Image View f x colon i d equals double quotes gray Image View double quotes Forward slash Right angled bracket. Line 17: blank. Line 18: Left angled bracket Label text equals double quotes ampersand l t semicolon hyphen darker lighter hyphen ampersand g t semicolon double quotes Forward slash Right angled bracket. Line 19: blank. Line 20: Left angled bracket Slider f x colon i d equals double quotes slider double quotes min equals double quotes 0 double quotes max equals double quotes dot 33 double quotes. Line 21: value equals double quotes dot 165 double quotes show Tick Marks equals double quotes true double quotes. Line 22: show Tick Labels equals double quotes true double quotes major Tick Unit equals double quotes 0 dot 055 double quotes Forward slash Right angled bracket. Line

23: blank. Line 24: Left angled bracket Forward slash V Box Right angled bracket.

Back to Example

The first 4 rows list X M L characters and the next 3 rows list F X M L characters. The row entries are as follows. Row 1. Character: less than. X M L Encoding Sequence: ampersand l t, semicolon. Row 2. Character: greater than. X M L Encoding Sequence: ampersand g t. Row 3. Character: ampersand. X M L Encoding Sequence: ampersand a m p, semicolon. Row 4. Character: double quotes. X M L Encoding Sequence: ampersand q u o t, semicolon. Row 5. Character: single quote. X M L Encoding Sequence: ampersand a p o s, semicolon. Row 5. Character: at symbol. F X M L Encoding Sequence: backslash at symbol. Row 6. Character: dollar sign. F X M L Encoding Sequence: backslash dollar sign. Row 7. Character: percentage sign. F X M L Encoding Sequence: backslash percentage sign.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, String, super, double, int, new, for, return. Line 1: Forward slash. Asterisk. Adjusts the gray value of an image. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java f x dot scene dot image dot Image, semicolon. Line 6: import java f x dot scene dot image dot Pixel

Reader, semicolon. Line 7: import java f x dot scene dot image dot Pixel Writer, semicolon. Line 8: import java f x dot scene dot image dot Writable Image, semicolon. Line 9: import java f x dot scene dot paint dot Color, semicolon. Line 10: blank. Line 11: public class Photo Grayer extends Image. Line 12: left brace. Line 13, indented once: Forward slash. 2 asterisks. constructor. Line 14, indented once: Asterisk. at symbol, param file file name of the image. Line 15, indented once: Asterisk. passes the file name to the Image class. Line 16, indented once: Asterisk. constructor. Line 17, indented once: Asterisk. Forward slash. Line 18, indented once: public Photo Grayer, left parentheses, String file, right parentheses. Line 19, indented once: left brace. Line 20, indented twice: super, left parentheses, file, right parentheses, semicolon. Line 21, indented once: right brace. Line 22: blank. Line 23, indented once: Forward slash. 2 asterisks. gray method. Line 24, indented once: Asterisk. at symbol, param c o e f f the multiplier to determine the. Line 25, indented once: Asterisk. color for each pixel. Line 26, indented once: Asterisk. at symbol, return a Writable Image. Line 27, indented once: Asterisk. Forward slash. Line 28, indented once: public Writable Image gray, left parentheses, double c o e f f, right parentheses. Line 29, indented once: left brace. Line 30, indented twice: int width, equals, left parentheses, int, right parentheses, get Width, left parentheses, right parentheses, semicolon. Line 31, indented twice: int height, equals, left parentheses, int, right parentheses, get Height, left parentheses, right parentheses, semicolon. Line 32, indented twice: Writable Image gray Image. Line 33, indented multiple times: equals, new, Writable Image, left parentheses, width, comma, height, right parentheses, semicolon. Line 34, indented twice: Pixel Writer p w, equals, gray Image dot get Pixel Writer, left parentheses, right parentheses, semicolon. Line 35, indented twice: Pixel Reader p r, equals, get Pixel Reader, left parentheses, right parentheses, semicolon. Line 36: blank. Line 37, indented twice: for, left parentheses, int x equals 0, semicolon, x less than width, semicolon, x plus, plus, right parentheses. Line 38, indented twice: left brace. Line 39, indented thrice: for, left parentheses, int y equals 0, semicolon, y less than height, semicolon, y plus, plus, right parentheses. Line 40, thrice: left brace. Line 41, indented 4 times: Color current Color, equals, p r dot get Color, left parentheses, x, comma, y, right parentheses, semicolon. Line 42, indented 4 times: double gray Value, equals, c o e f f asterisk. Line 43, indented multiple times: left parentheses, current Color dot get Red, left parentheses, right parentheses, plus, current Color dot get Green, left parentheses, right parentheses. Line 44, indented multiple times: plus, current Color dot get Blue, left parentheses, right parentheses, right parentheses, semicolon. Line 45, indented 4 times: current Color, equals, Color dot gray, left parentheses, gray Value, right parentheses, semicolon. Line 46, indented 4 times: p w dot set Color, left parentheses, x, comma, y, comma, current Color, right parentheses, semicolon. Line 47, indented thrice: right brace. Line 48, indented twice: right brace. Line 49, indented twice: return gray Image, semicolon. Line 50, indented once: right brace. Line 51: right brace.

Back to Example

The methods are in the java f x dot scene dot image package. The constructor is as follows: Image, left parentheses, String U R L, right parentheses, constructs and Image from the file named in U R L. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: Pixel Reader. Method name and argument list: get Pixel Reader, left parentheses, right parentheses, returns a Pixel Reader object for accessing the pixels in the Image. Row 2. Return value: double. Method name and argument list: get Height, left parentheses, right parentheses, returns the height of the image in pixels. Row 3. Return value: double. Method name and argument list: get Width, left parentheses, right parentheses, returns the width of the image in pixels.

Back to Table

The method is in the java f x dot scene dot image package. The constructor is as follows: Writable Image, left parentheses, int width, comma, int height, right parentheses, constructs an empty Writable Image of the specified width and height. The table has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Return value: Pixel Writer. Method name and argument list: get Pixel Writer, left parentheses, right parentheses, returns a Pixel Writer object for writing the pixels in the image.

Back to Table

The method is in the java f x dot scene dot image package. The table has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Return value: Color. Method name and argument list: get Color, left parentheses, int x, comma, int y, right parentheses, returns the Color of the pixel at coordinate, left parentheses, x, comma, y, right parentheses, in the image.

Back to Table

The method is in the java f x dot scene dot image package. The table has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Return value: void. Method name and argument list: set Color, left parentheses, int x, comma, int y, comma, Color c, right parentheses, sets the pixel at coordinate, left parentheses, x, comma, y, right parentheses, to the color c.

Back to Table

The methods are in the java f x dot scene dot paint package. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: double. Method name and argument list: get Red, left parentheses, right parentheses, returns the red component of the color in the range 0.0 to 1.0. Row 2. Return value: double. Method name and argument list: get Green, left parentheses, right parentheses, returns the green component of the color in the range 0.0 to 1.0. Row 3. Return value: double. Method name and argument list: get Blue, left parentheses, right parentheses, returns the blue component of the color in the range 0.0 to 1.0. Row 4. Return value: Color. Method name and argument list: gray, left parentheses, double value, right parentheses, returns a Color object where the red, green, and blue components are set to value, which can range from 0.0 to 1.0.

Back to Table

Program code. The code contains the following keywords: import, public, class, private, void, new, extends. Line 1: import java f x dot beans dot value dot asterisk, semicolon. Line 2: import java f x dot event dot asterisk, semicolon. Line 3: import java f x dot f x m l dot F X M L, semicolon. Line 4: import java f x dot scene dot control dot asterisk, semicolon. Line 5: import java f x dot scene dot image dot asterisk, semicolon. Line 6: blank. Line 7: public class Photo Grayer Controller. Line 8: left brace. Line 9, indented once: private Photo Grayer photo Grayer, semicolon. Line 10: blank. Line 11, indented once: at symbol, F X M L private Image View gray Image View, semicolon. Line 12, indented once: at symbol, F X M L private Slider slider, semicolon. Line 13: blank. Line 14, indented once: public void initialize, left parentheses, right parentheses. Line 15, indented once: left brace. Line 16, indented twice: photo Grayer, equals, new Photo Grayer, left parentheses, double quotes, sushi dot j p g, double quotes, right parentheses, semicolon. Line 17: blank. Line 18, indented twice: 2 forward slashes, initialize gray Image View. Line 19, indented twice: Image gray Image, equals, photo Grayer dot gray, left parentheses, slider dot get Value, left parentheses, right parentheses, right parentheses, semicolon. Line 20, indented twice: gray Image View dot set Image, left parentheses, gray Image, right parentheses, semicolon. Line 21: blank. Line 22, indented twice: 2 forward slashes, set up event handling for slider. Line 23, indented twice: Slider Handler s h, equals, new, Slider Handler, left parentheses, right parentheses, semicolon. Line 24, indented twice: slider dot value Property, left parentheses, right parentheses, dot, add Listener, left parentheses, s h, right parentheses, semicolon. Line 25, indented once: right brace. Line 26: blank. Line 27, indented once: private class Slider Handler. Line 28, indented multiple times: implements Change Listener, left angled bracket, Number, right angled bracket. Line 29, indented once: left brace. Line 30, indented twice: at symbol, Override. Line 31, indented twice: public void changed, left parentheses, Observable Value, left angled bracket, question mark, extends

Number, right angled bracket, o, comma. Line 32, indented multiple times: Number old Value, comma, Number new Value, right parentheses. Line 33, indented twice: left brace. Line 34, indented thrice: 2 forward slashes, update gray Image View. Line 35, indented thrice: gray Image View dot set Image, left parentheses. Line 36, indented multiple times: photo Grayer dot gray, left parentheses, new Value dot double Value, left parentheses, right parentheses, right parentheses, right parentheses, semicolon. Line 37, indented twice: right brace. Line 38, indented once: right brace. Line 39: right brace.

Back to Example

The methods are in the java f x dot scene dot control package. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: double. Method name and argument list: get Value, left parentheses, right parentheses, returns the current value of the Slider as a double. Row 2. Return value: Double Property. Method name and argument list: value Property, left parentheses, right parentheses, returns the current value of the Slider as a Double Property which is a Property wrapper class for a double, and implements the Observable Value, left angled bracket, Number, right angled bracket, interface.

Back to Table

The method is in the java f x dot beans dot value package. The row entry is as follows. Return value: void. Method name and argument list: changed, left parentheses, Observable Value, left angled bracket, question mark, extends uppercase t, right angled bracket, observable, comma, uppercase t, old Value, comma, uppercase t, new Value, right parentheses, The Change Listener must implement this method to handle changes to the Observable Value.

Back to Table

The method is in the java dot lang package. The table has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Return value: double. Method name and argument list: double Value, left parentheses, right parentheses, abstract method in the Number class overridden by the Double class, semicolon, it returns the value of a Double object as a double.

Back to Table

The methods are in the java f x dot beans dot value package. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: add Listener, left parentheses, Change Listener, left angled bracket, question mark, super, uppercase t, right angled bracket, listener, right parentheses, adds a Change Listener whose code will execute whenever the value of the Observable Value changes. Row 2. Return value: uppercase t. Method name and argument list: get Value, left parentheses, right parentheses, returns the current value of the Observable Value object. Row 3. Return value: void. Method name and argument list: remove Listener, left parentheses, Change Listener, left angled bracket, question mark, super, uppercase t, right angled bracket, listener, right parentheses, removes the listener so that it is no longer notified of changes to the value of the Observable Value.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, void, try, new, catch, Exception, static, String. Line 1: Forward slash. Asterisk. Slider demo. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot net dot U R L, semicolon. Line 6: import java f x dot application dot Application, semicolon. Line 7: import java f x dot f x m l dot F X M L Loader, semicolon. Line 8: import java f x dot scene dot layout dot V Box, semicolon. Line 9: import java f x dot scene dot Scene, semicolon. Line 10: import java f x dot stage dot Stage, semicolon. Line 11: blank. Line 12: public class Photo Graying extends Application. Line 13: left brace. Line 14, indented once: at symbol, Override. Line 15, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 16, indented once: left brace. Line 17, indented twice: try. Line 18, indented twice: left brace. Line 19, indented thrice: U R L u r l, equals. Line 20, indented 4 times: get Class, left parentheses, right parentheses, dot get Resource, left parentheses, double quotes, f x m l underscore photo underscore grayer dot f x m l, double quotes, right parentheses, semicolon. Line 21, indented thrice: V Box root, equals, F X M L Loader dot load, left parentheses, u r l, right parentheses, semicolon. Line 22, indented thrice: Scene scene, equals, new Scene, left parentheses, root, comma, 350, comma, 320, right parentheses, semicolon. Line 23, indented thrice: stage dot set Title, left parentheses, double quotes, Graying a photo, double quotes, right parentheses, semicolon. Line 24, indented thrice: stage dot set Scene, left parentheses, scene, right parentheses, semicolon. Line 25, indented thrice: stage dot show, left parentheses, right parentheses, semicolon. Line 26, indented twice: right brace. Line 27, indented twice: catch, left parentheses, Exception e, right parentheses. Line 28, indented twice: left brace. Line 29, indented thrice: e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 30, indented twice: right brace. Line 31, indented once: right brace. Line 32: blank. Line 33, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 34, indented once: left brace. Line 35, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 36, indented once: right brace. Line 37: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, private, static, char, return, double, if, else. Line 1: Forward slash. Asterisk. Board Game class. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java f x dot scene dot paint dot Color semicolon. Line 6: blank. Line 7: public class Board Game. Line 8: left brace. Line 9: private static char left square bracket, right square bracket hex Digits equals. Line 10: left brace single quote 0 single quote comma single quote 1 single quote comma single quote 2 single quote comma single quote 3 single quote comma single quote 4 single quote comma single quote 5 single quote comma single quote 6 single quote comma single quote 7 single quote comma single quote 8 single quote comma single quote 9 single quote comma. Line 11: single quote A single quote comma single quote B single quote comma single quote C single quote comma single quote D single quote comma single quote E single quote comma single quote F single quote right brace semicolon. Line 12: private int left square bracket, right square bracket rows semicolon. Line 13: private char left square bracket, right square bracket columns semicolon. Line 14: private Color left square bracket, right square bracket colors semicolon Forward slash Forward slash to color the rows and columns. Line 15: blank. Line 16: Forward slash. Asterisk. Asterisk Overloaded Constructor. Line 17: Asterisk at symbol param new Rows the row numbers. Line 18: Asterisk at symbol param new Columns the column letters. Line 19: Asterisk at symbol param new Colors the colors. Line 20: Asterisk. Forward slash. Line 21: public Board Game left parentheses int left square bracket, right square bracket new Rows comma char left square bracket, right square bracket new Columns comma. Line 22: Color left square bracket, right square bracket new Colors right parentheses. Line 23: left brace. Line 24: rows equals new Rows semicolon. Line 25: columns equals new Columns semicolon. Line 26: colors equals new Colors semicolon. Line 27: right brace. Line 28: blank. Line 29: Forward slash. Asterisk. Asterisk get Square Text method. Line 30: Asterisk at symbol param row the row index. Line 31: Asterisk at symbol param column the column index. Line 32: Asterisk at symbol return a String comma a concatenation of columns left square bracket column right square bracket and rows left square bracket row right square bracket. Line 33: Asterisk. Forward slash. Line 34: public String get Square Text left parentheses int row comma int column right parentheses. Line 35: left brace. Line 36: return String dot value Of left parentheses columns left square bracket column right square bracket right parentheses plus rows left square bracket row right square bracket semicolon. Line 37: right brace. Line 38: blank. Line 39: Forward slash. Asterisk. Asterisk get Square Color method. Line 40: Asterisk at symbol param row the row index. Line 41: Asterisk at symbol param column the column index. Line 42: Asterisk at symbol return a String comma a hex representation of. Line 43: Asterisk colors left square bracket left parentheses row plus col right parentheses percentage sign colors dot length right square bracket. Line 44: Asterisk. Forward slash. Line 45: public String get Square Color left parentheses int row comma int col right parentheses. Line 46: left brace. Line 47: Color square Color equals colors left square bracket left parentheses row plus col right parentheses percentage sign colors dot length right square bracket semicolon. Line 48: return to Hex String left parentheses square Color right parentheses semicolon. Line 49: right brace. Line 50: blank. Line 51: Forward slash. Asterisk. Asterisk to Hex utility method. Line 52: Asterisk at symbol param color Intensity a double between 0 and 1 inclusive. Line 53: Asterisk at symbol return a String comma the hex representation of color Intensity times 255. Line 54: Asterisk. Forward slash. Line 55: public String to Hex left parentheses double color Intensity right parentheses. Line 56: left brace. Line 57: int color Intensity As Int equals left parentheses int right parentheses left parentheses color Intensity Asterisk 255 right parentheses semicolon. Line 58: if left parentheses color Intensity As Int Right angled bracket 255 right parentheses. Line 59: color Intensity As Int equals 255semicolon. Line 60: else if left parentheses color Intensity As Int Left angled bracket 0 right parentheses. Line 61: color Intensity As Int equals 0 semicolon. Line 62: blank. Line 63: int first Digit equals color Intensity As Int Forward slash 16 semicolon. Line 64: int second Digit equals color Intensity As Int percentage sign 16 semicolon. Line 65: blank. Line 66: return String dot value Of left parentheses hex Digits left square bracket first Digit right square bracket right parentheses plus hex Digits left square bracket second Digit right square bracket semicolon. Line 67: right brace. Line 68: blank. Line 69: Forward slash. Asterisk. Asterisk to Hex String method. Line 70: Asterisk at symbol param color a Color. Line 71: Asterisk at symbol return a String comma a hex representation of color. Line 72: Asterisk. Forward slash. Line 73: public String to Hex String left parentheses Color color right parentheses. Line 74: left brace. Line 75: String color Text equals double quotes hash double quotes plus to Hex left parentheses color dot get Red left parentheses right parentheses right parentheses. Line 76: plus to Hex left parentheses color dot get Green left parentheses right parentheses right parentheses. Line 77: plus to Hex left parentheses color dot get Blue left parentheses right parentheses right parentheses semicolon. Line 78: return color Text semicolon. Line 79: right brace. Line 80: blank. Line 81: Forward slash. Asterisk. Asterisk get Number Of Rows method. Line 82: Asterisk at symbol return the number of rows. Line 83: Asterisk. Forward slash. Line 84: public int get Number Of Rows left parentheses right parentheses. Line 85: left brace. Line 86: return rows dot length semicolon. Line 87: right brace. Line 88: blank. Line 89: Forward slash. Asterisk. Asterisk get Number Of Columns method. Line 90: Asterisk at symbol return the number of columns. Line 91: Asterisk. Forward slash. Line

92: public int get Number Of Columns left parentheses right parentheses. Line 93: left brace. Line 94: return columns dot length semicolon. Line 95: right brace. Line 96: right brace.

Back to Example

The methods are in the java f x dot scene dot layout package. The row entries are as follows. Row 1. Return value: void. Method name and argument list: add, left parentheses, Node child, comma, int column Index, comma, int row Index, right parentheses, adds child to this Grid Pane at row row Index and column column Index. Row 2. Return value: Observable List, left angled bracket, Row Constraints, right angled bracket. Method name and argument list: get Row Constraints, left parentheses, right parentheses, returns a list of row constraints for this Grid Pane. Row 3. Return value: Observable List, left angled bracket, Column Constraints, right angled bracket. Method name and argument list: get Column Constraints, left parentheses, right parentheses, returns a list of column constraints for this Grid Pane.

Back to Table

The methods are in the java dot util package. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: boolean. Method name and argument list: add, left parentheses, uppercase e, lowercase e, right parentheses, appends lowercase e to the end of the list. Row 2. Return value: void. Method name and argument list: clear, left parentheses, right parentheses, removes all the elements from the list.

Back to Table

The method is in the java f x dot scene dot layout package. The table has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Return value: void. Method name and argument list: set Percent Height, left parentheses, double value, right parentheses, sets the height of a row as a percentage of the total height of the Grid Pane.

Back to Table

The method is in the java f x dot scene dot layout package. The table has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Return value: void. Method name and argument list: set Percent Width, left parentheses, double value, right parentheses, sets the width of a column as a percentage of the total width of the Grid Pane.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, private, super, int, new, for, void, if. Line 1: Forward slash. Asterisk. Using Grid Pane to organize our window. Line 2: Anderson comma

Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java f x dot event dot Action Event semicolon. Line 6: import java f x dot event dot Event Handler semicolon. Line 7: import java f x dot scene dot control dot Button semicolon. Line 8: import java f x dot scene dot layout dot Asterisk semicolon. Line 9: blank. Line 10: public class Board View extends Grid Pane. Line 11: left brace. Line 12: private Board Game game semicolon. Line 13: private Button left square bracket, right square bracket left square bracket, right square bracket squares semicolon. Line 14: blank. Line 15: public Board View left parentheses Board Game new Game right parentheses. Line 16: left brace. Line 17: super left parentheses right parentheses semicolon. Line 18: game equals new Game semicolon. Line 19: blank. Line 20: Forward slash Forward slash set up grid according to Model. Line 21: int rows equals game dot get Number Of Rows left parentheses right parentheses semicolon. Line 22: int columns equals game dot get Number Of Columns left parentheses right parentheses semicolon. Line 23: Row Constraints row equals new Row Constraints left parentheses right parentheses semicolon. Line 24: row dot set Percent Height left parentheses 100 dot 0 Forward slash rows right parentheses semicolon. Line 25: Column Constraints c o l equals new Column Constraints left parentheses right parentheses semicolon. Line 26: col dot set Percent Width left parentheses 100 dot 0 Forward slash columns right parentheses semicolon. Line 27: for left parentheses int i equals 0semicolon i Left angled bracket rows semicolon i plus plus right parentheses. Line 28: get Row Constraints left parentheses right parentheses dot add left parentheses row right parentheses semicolon. Line 29: for left parentheses int j equals 0 semicolon j Left angled bracket columns semicolon j plus plus right parentheses. Line 30: get Column Constraints left parentheses right parentheses dot add left parentheses col right parentheses semicolon. Line 31: blank. Line 32: squares equals new Button left square bracket rows right square bracket left square bracket columns right square bracket semicolon. Line 33: Button Handler b h equals new Button Handler left parentheses right parentheses semicolon. Line 34: blank. Line 35: for left parentheses int i equals 0 semicolon i Left angled bracket rows semicolon i plus plus right parentheses. Line 36: left brace. Line 37: for left parentheses int j equals 0 semicolon j Left angled bracket columns semicolon j plus plus right parentheses. Line 38: left brace. Line 39: Forward slash Forward slash instantiate the button with no text. Line 40: squares left square bracket i right square bracket left square bracket j right square bracket equals new Button left parentheses right parentheses semicolon. Line 41: blank. Line 42: Forward slash Forward slash color the button. Line 43: squares left square bracket i right square bracket left square bracket j right square bracket dot set Style left parentheses double quotes hyphen f x hyphen background hyphen color colon double quotes. Line 44: plus game dot get Square Color left parentheses i comma j right parentheses right parentheses semicolon. Line 45: blank. Line 46: Forward slash Forward slash add the button. Line 47: add left parentheses squares left square bracket i right square bracket left square bracket j right square bracket comma j comma i right parentheses semicolon. Line 48: blank. Line 49: Forward slash Forward slash make button fill up available width and height. Line 50: squares left square bracket i right square bracket left square bracket j right square bracket dot set Max Width left parentheses Double dot MAX underscore VALUE right parentheses semicolon. Line 51: squares left square bracket i right square bracket left square bracket j right square bracket dot set Max Height left parentheses Double dot MAX underscore VALUE right parentheses semicolon. Line 52: blank. Line 53: Forward slash Forward slash register listener on button. Line 54: squares left square bracket i right square bracket left square bracket j right square bracket dot set On Action left parentheses b h right parentheses semicolon. Line 55: right brace. Line 56: right brace. Line 57: right brace. Line 58: blank. Line 59: Forward slash Forward slash private inner class event handler. Line 60: private class Button Handler implements Event Handler Left angled bracket Action Event Right angled bracket. Line 61: left brace. Line 62: at symbol Override. Line 63: public void handle left parentheses Action Event event right parentheses. Line 64: left brace. Line 65: for left parentheses int i equals 0semicolon i Left angled bracket squares dot length semicolon i plus plus right parentheses. Line 66:

left brace. Line 67: for left parentheses int j equals 0 semicolon j Left angled bracket squares left square bracket i right square bracket dot length semicolon j plus plus right parentheses. Line 68: left brace. Line 69: if left parentheses event dot get Source left parentheses right parentheses equals equals squares left square bracket i right square bracket left square bracket j right square bracket right parentheses. Line 70: left brace. Line 71: squares left square bracket i right square bracket left square bracket j right square bracket dot set Text left parentheses game dot get Square Text left parentheses i comma j right parentheses right parentheses semicolon. Line 72: return semicolon. Line 73: right brace. Line 74: right brace. Line 75: right brace. Line 76: right brace. Line 77: right brace. Line 78: right brace.

Back to Example

The methods are in the java f x dot scene dot control package. The constructor is as follows: Button, left parentheses, right parentheses, creates an empty Button. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: set Style, left parentheses, String style, right parentheses, sets the value of the style property of this Button to style, which is expected to be a C S S like string. This method is inherited from Node. Row 2. Return value: void. Method name and argument list: set Max Width, left parentheses, double value, right parentheses, sets the value of the max Width property of this Button to value. This method is inherited from Region. Row 3. Return value: void. Method name and argument list: set Max Height, left parentheses, double value, right parentheses, sets the value of the max Height property of this Button to value. This method is inherited from Region. Row 4. Return value: void. Method name and argument list: set On Action, left parentheses, Event Handler, left angled bracket, Action Event, right angled bracket, handler, right parentheses, registers handler as the event handler for this Button. When the button is clicked, the handle method of handler's class will be called automatically. This method is inherited from Button Base.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, void, char, int, new, static, String. Line 1: Forward slash. Asterisk. Chess Board class. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java f x dot application dot Application, semicolon. Line 6: import java f x dot scene dot paint dot Color, semicolon. Line 7: import java f x dot scene dot Scene, semicolon. Line 8: import java f x dot stage dot Stage, semicolon. Line 9: blank. Line 10: public class Chess Board extends Application. Line 11: left brace. Line 12, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 13, indented once: left brace. Line 14, indented twice: char, left square bracket, right square bracket, letters, equals, left brace, single quote, a, single quote, comma, single quote, b, single quote, comma, single quote, c, single quote, comma, single quote, d, single quote, comma, single quote, e, single quote, comma, single quote, f, single quote, comma, single quote, g, single quote, comma, single quote, h, single quote, right brace, semicolon. Line 15, indented twice: int, left square bracket, right square bracket, digits, equals, left brace, 8, comma, 7, comma, 6, comma, 5, comma, 3, comma, 2, comma, 1, right brace, semicolon. Line 16, indented twice: Color, left square bracket, right square bracket, board Colors, equals, left brace, Color dot WHITE, comma, Color dot RED, right brace, semicolon. Line 17: blank. Line 18, indented twice: Board Game game, equals, new Board Game, left parentheses, digits, comma, letters, comma, board Colors, right parentheses, semicolon. Line 19, indented twice: Board View view, equals, new, Board View, left parentheses, game, right parentheses, semicolon. Line 20: blank. Line 21, indented twice: Scene scene, equals, new Scene, left parentheses, view, comma, 450, comma, 450, right parentheses, semicolon. Line 22, indented twice: stage dot set Scene, left parentheses, scene, right parentheses, semicolon. Line 23, indented twice: stage dot set Title, left parentheses, double quotes, Click a square to reveal its position, double quotes, right parentheses, semicolon. Line 24, indented twice: stage dot show, left parentheses, right parentheses, semicolon. Line 25, indented once: right brace. Line 26: blank. Line 27, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 28, indented once; left brace. Line 29, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 30, indented once: right brace. Line 31: right brace.

Back to Example

Program code. The code contains the following keywords: public, class, private, int, String, void, if, else, new, for, return, true, false, boolean. Line 1: Forward slash. Asterisk. Asterisk Tile Puzzle class. Line 2: Asterisk Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: public class Tile Puzzle. Line 6: left brace. Line 7: private String left square bracket, right square bracket left square bracket, right square bracket tiles semicolon. Line 8: private int side semicolon Forward slash Forward slash grid size. Line 9: private int empty Row semicolon. Line 10: private int empty Col semicolon. Line 11: blank. Line 12: Forward slash. Asterisk. Asterisk constructor. Line 13: Asterisk at symbol param new Side grid size. Line 14: Asterisk. Forward slash. Line 15: public Tile Puzzle left parentheses int new Side right parentheses. Line 16: left brace. Line 17: set Up Game left parentheses new Side right parentheses semicolon. Line 18: right brace. Line 19: blank. Line 20: Forward slash. Asterisk. Asterisk set Up Game. Line 21: Asterisk at symbol param new Side grid size. Line 22: Asterisk. Forward slash. Line 23: public void set Up Game left parentheses int new Side right parentheses. Line 24: left brace. Line 25: if left parentheses new Side Left angled bracket 3 right parentheses. Line 26: side equals 3 semicolon. Line 27: else. Line 28: side equals new Side semicolon. Line 29: empty Row equals side hyphen 1 semicolon. Line 30: empty Col equals side hyphen 1 semicolon. Line 31: tiles equals new String left square bracket side right square bracket left square bracket side right square bracket semicolon. Line 32: blank. Line 33: Forward slash Forward slash initialize tiles. Line 34: for left parentheses int i equals 0 semicolon i Left angled bracket side semicolon i plus plus right parentheses. Line 35: left brace. Line 36: for left parentheses int j equals 0semicolon j Left angled bracket side semicolon j plus plus right parentheses. Line 37: left brace. Line 38: tiles left square bracket i right square bracket left square bracket j right square bracket equals String dot value Of left parentheses left parentheses side Asterisk side right parentheses. Line 39: hyphen left parentheses side Asterisk i plus j plus 1 right parentheses right parentheses semicolon. Line 40: right brace. Line 41: right brace. Line 42: Forward slash Forward slash set empty cell label to blank. Line 43: tiles left square bracket side hyphen 1 right square bracket left square bracket side hyphen 1 right square bracket equals double quotes double quotes semicolon. Line 44: right brace. Line 45: blank. Line 46: Forward slash. Asterisk. Asterisk get Side. Line 47: Asterisk at symbol return side. Line 48: Asterisk. Forward slash. Line 49: public int get Side left parentheses right parentheses. Line 50: left brace. Line 51: return side semicolon. Line 52: right brace. Line 53: blank. Line 54: Forward slash. Asterisk. Asterisk get Tiles. Line 55: Asterisk at symbol return a copy of tiles. Line 56: Asterisk. Forward slash. Line 57: public String left square bracket, right square bracket left square bracket, right square bracket get Tiles left parentheses right parentheses. Line 58: left brace. Line 59: String left square bracket, right square bracket left square bracket, right square bracket copy Of Tiles equals new String left square bracket side right square bracket left square bracket side right square bracket semicolon. Line 60: blank. Line 61: for left parentheses int i equals 0 semicolon i Left angled bracket side semicolon i plus plus right parentheses. Line 62: left brace. Line 63: for left parentheses int j equals 0 semicolon j Left angled bracket side semicolon j plus plus right parentheses. Line 64: left brace. Line 65: copy Of Tiles left square bracket i right square bracket left square bracket j right square bracket equals tiles left square bracket i right square bracket left square bracket j right square bracket semicolon. Line 66: right brace. Line 67: right brace. Line 68: return copy Of Tiles semicolon. Line 69: right brace. Line 70: blank. Line 71: Forward slash. Asterisk. Asterisk try To Play. Line 72: Asterisk enable play if play is legal. Line 73: Asterisk at symbol return true if the play is legal comma false otherwise. Line 74: Asterisk. Forward slash. Line 75: public boolean try To Play left parentheses int row comma int col right parentheses. Line 76: left brace. Line 77: if left parentheses possible To Play left parentheses row comma col right parentheses right parentheses. Line 78: left brace. Line 79: Forward slash Forward slash play colon switch empty String and tile label at row comma col. Line 80: tiles left square bracket empty Row right square bracket left square bracket empty Col right square bracket equals tiles left square bracket row right square bracket left square bracket col right square bracket semicolon. Line 81: tiles left square bracket row right square bracket left square bracket col right square bracket equals double quotes double quotes semicolon. Line 82: Forward slash Forward slash update empty Row and empty Col. Line 83: empty Row equals row semicolon. Line 84: empty Col equals col semicolon. Line 85: return true semicolon. Line 86: right brace. Line 87: else. Line 88: return false semicolon. Line 89: right brace. Line 90: blank. Line 91: Forward slash. Asterisk. Asterisk possible To Play. Line 92: Asterisk at symbol return true if the play is legal comma false otherwise. Line 93: Asterisk. Forward slash. Line 94: public boolean possible To Play left parentheses int row comma int col right parentheses. Line 95: left brace. Line 96: if left parentheses left parentheses col equals equals empty Col ampersand ampersand Math dot a b s left parentheses row hyphen empty Row right parentheses equals equals 1 right parentheses. Line 97: 2 pipes left parentheses row equals equals empty Row ampersand ampersand Math dot abs left parentheses col hyphen empty Col right parentheses equals equals 1 right parentheses right parentheses. Line 98: return true semicolon. Line 99: else. Line 100: return false semicolon. Line 101: right brace. Line 102: blank. Line 103: Forward slash. Asterisk. Asterisk won. Line 104: Asterisk at symbol return true if correct tile order comma false otherwise. Line 105: Asterisk. Forward slash. Line 106: public boolean won left parentheses right parentheses. Line 107: left brace. Line 108: for left parentheses int i equals 0semicolon i Left angled bracket side semicolon i plus plus right parentheses. Line 109: left brace. Line 110: for left parentheses int j equals 0 semicolon j Left angled bracket side semicolon j plus plus right parentheses. Line 111: left brace. Line 112: if left parentheses exclamation mark left parentheses tiles left square bracket i right square bracket left square bracket j right square bracket dot equals left parentheses. Line 113: String dot value Of left parentheses i Asterisk side plus j plus 1 right parentheses right parentheses right parentheses. Line 114: ampersand ampersand left parentheses i exclamation mark equals side hyphen 1 2 pipes j exclamation mark equals side hyphen 1 right parentheses right parentheses. Line 115: return false semicolon. Line 116: right brace. Line 117: right brace. Line 118: return true semicolon. Line 119: right brace. Line 120: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, private, int, String, return, void, if, this. Line 1: Forward slash. Asterisk. Asterisk Grid

Button class. Line 2: Asterisk Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java f x dot scene dot control dot Button semicolon. Line 6: blank. Line 7: public class Grid Button extends Button. Line 8: left brace. Line 9: private int row semicolon. Line 10: private int column semicolon. Line 11: blank. Line 12: Forward slash. Asterisk. Asterisk Constructor. Line 13: Asterisk at symbol param title text for button. Line 14: Asterisk at symbol param new Row row. Line 15: Asterisk at symbol param new Column column. Line 16: Asterisk. Forward slash. Line 17: public Grid Button left parentheses String title comma int new Row comma int new Column right parentheses. Line 18: left brace. Line 19: super left parentheses title right parentheses semicolon. Line 20: set Row left parentheses new Row right parentheses semicolon. Line 21: set Column left parentheses new Column right parentheses semicolon. Line 22: right brace. Line 23: blank. Line 24: Forward slash. Asterisk. Asterisk get Row method comma accessor for row. Line 25: Asterisk at symbol return row. Line 26: Asterisk. Forward slash. Line 27: public int get Row left parentheses right parentheses. Line 28: left brace. Line 29: return row semicolon. Line 30: right brace. Line 31: blank. Line 32: Forward slash. Asterisk. Asterisk get Column method comma accessor for column. Line 33: Asterisk at symbol return column. Line 34: Asterisk. Forward slash. Line 35: public int get Column left parentheses right parentheses. Line 36: left brace. Line 37: return column semicolon. Line 38: right brace. Line

39: blank. Line 40: Forward slash. Asterisk. Asterisk set Row method comma mutator for row. Line 41: Asterisk at symbol param row comma new value for row. Line 42: Asterisk. Forward slash. Line 43: public void set Row left parentheses int row right parentheses. Line 44: left brace. Line 45: if left parentheses row Right angled bracket equals 0 right parentheses. Line 46: this dot row equals row semicolon. Line 47: right brace. Line 48: blank. Line 49: Forward slash. Asterisk. Asterisk set Column method. Line 50: Asterisk at symbol param column comma new value for column. Line 51: Asterisk. Forward slash. Line 52: public void set Column left parentheses int column right parentheses. Line 53: left brace. Line 54: if left parentheses column Right angled bracket equals 0 right parentheses. Line 55: this dot column equals column semicolon. Line 56: right brace. Line 57: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, private, for, int, new, void, if, private. Line 1: Forward slash. Asterisk. Using Grid Pane dynamically. Line 2: Anderson comma Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java dot util dot Random semicolon. Line 6: import java f x dot event dot Action Event semicolon. Line 7: import java f x dot event dot Event Handler semicolon. Line 8: import java f x dot scene dot control dot Alert dot Alert Type semicolon. Line 9: import java f x dot scene dot control dot Asterisk semicolon. Line 10: import java f x dot scene dot layout dot Asterisk semicolon. Line 11: blank. Line 12: public class Tile Puzzle View Controller extends Grid Pane. Line 13: left brace. Line 14: private Grid Button left square bracket, right square bracket left square bracket, right square bracket, squares semicolon. Line 15: private Tile Puzzle game semicolon Forward slash Forward slash the tile puzzle game. Line 16: blank. Line 17: public Tile Puzzle View Controller left parentheses Tile Puzzle new Game right parentheses. Line 18: left brace. Line 19: super left parentheses right parentheses semicolon. Line 20: game equals new Game semicolon. Line 21: set Up Game G U I left parentheses right parentheses semicolon. Line 22: right brace. Line 23: blank. Line 24: public void set Up Game G U I left parentheses right parentheses. Line 25: left brace. Line 26: Forward slash Forward slash remove all components and constraints. Line 27: get Children left parentheses right parentheses dot clear left parentheses right parentheses semicolon. Line 28: get Row Constraints left parentheses right parentheses dot clear left parentheses right parentheses semicolon. Line 29: get Column Constraints left parentheses right parentheses dot clear left parentheses right parentheses semicolon. Line 30: blank. Line 31: Forward slash Forward slash set up grid constraints. Line 32: Row Constraints row equals new Row Constraints left parentheses right parentheses semicolon. Line 33: row dot set Percent Height left parentheses 100 dot 0 Forward slash game dot get Side left parentheses right parentheses right parentheses semicolon. Line 34: Column Constraints col equals new Column Constraints left parentheses right parentheses semicolon. Line 35: col dot set Percent Width left parentheses 100 dot 0 Forward slash game dot get Side left parentheses right parentheses right parentheses semicolon. Line 36: for left parentheses int i equals 0semicolon i Left angled bracket game dot get Side left parentheses right parentheses semicolon i plus plus right parentheses. Line 37: get Row Constraints left parentheses right parentheses dot add left parentheses row right parentheses semicolon. Line 38: for left parentheses int j equals 0 semicolon j Left angled bracket game dot get Side left parentheses right parentheses semicolon j plus plus right parentheses. Line 39: get Column Constraints left parentheses right parentheses dot add All left parentheses col right parentheses semicolon. Line 40: blank. Line 41: squares equals new Grid Button left square bracket game dot get Side left parentheses right parentheses right square bracket left square bracket game dot get Side left parentheses right parentheses right square bracket semicolon. Line 42: Button Handler b h equals new Button Handler left parentheses right parentheses semicolon. Line 43: blank. Line 44: for left parentheses int i equals 0 semicolon i Left angled bracket game dot get Side left parentheses right parentheses semicolon i plus plus right parentheses. Line 45: left brace. Line 46: for left parentheses int j equals 0 semicolon j Left angled bracket game dot get Side left parentheses right parentheses semicolon j plus plus right parentheses. Line 47: left brace. Line 48: squares left square bracket i right square bracket left square bracket j right square bracket equals new Grid Button left parentheses game dot get Tiles left parentheses right parentheses left square bracket i right square bracket left square bracket j right square bracket comma i comma j right parentheses semicolon. Line 49: blank. Line 50: Forward slash Forward slash add the button. Line 51: add left parentheses squares left square bracket i right square bracket left square bracket j right square bracket comma j comma i right parentheses semicolon. Line 52: blank. Line 53: Forward slash Forward slash make button fill up available width and height. Line 54: squares left square bracket i right square bracket left square bracket j right square bracket dot set Max Width left parentheses Double dot MAX underscore VALUE right parentheses semicolon. Line 55: squares left square bracket i right square bracket left square bracket j right square bracket dot set Max Height left parentheses Double dot MAX underscore VALUE right parentheses semicolon. Line 56: blank. Line 57: Forward slash Forward slash register listener on button. Line 58: squares left square bracket i right square bracket left square bracket j right square bracket dot set On Action left parentheses b h right parentheses semicolon. Line 59: right brace. Line 60: right brace. Line 61: right brace. Line 62: blank. Line 63: public void update left parentheses right parentheses. Line 64: left brace. Line 65: for left parentheses int i equals 0semicolon i Left angled bracket game dot get Side left parentheses right parentheses semicolon i plus plus right parentheses. Line 66: for left parentheses int j equals 0 semicolon j Left angled bracket game dot get Side left parentheses right parentheses semicolon j plus plus right parentheses. Line 67: squares left square bracket i right square bracket left square bracket j right square bracket dot set Text left parentheses game dot get Tiles left parentheses right parentheses left square bracket i right square bracket left square bracket j right square bracket right parentheses semicolon. Line 68: blank. Line 69: if left parentheses game dot won left parentheses right parentheses right parentheses. Line 70: left brace. Line 71: show Message Dialog left parentheses double quotes Congratulations double quotes comma. Line 72: double quotes You won backslash n Setting up a new game double quotes right parentheses semicolon. Line 73: Random random equals new Random left parentheses right parentheses semicolon. Line 74: int side Of Puzzle equals 3 plus random dot next Int left parentheses 4 right parentheses semicolon. Line 75: game dot set Up Game left parentheses side Of Puzzle right parentheses semicolon. Line 76: set Up Game G U I left parentheses right parentheses semicolon. Line 77: right brace. Line 78: right brace. Line 79: blank. Line 80: public void show Message Dialog left parentheses String title comma String message right parentheses. Line 81: left brace. Line 82: Alert alert equals new Alert left parentheses Alert Type dot INFORMATION right parentheses semicolon. Line 83: alert dot set Title left parentheses title right parentheses semicolon. Line 84: alert dot set Header Text left parentheses double quotes double quotes right parentheses semicolon. Line 85: alert dot set Content Text left parentheses message right parentheses semicolon. Line 86: alert dot show And Wait left parentheses right parentheses semicolon. Line 87: right brace. Line 88: blank. Line 89: Forward slash Forward slash private inner class event handler. Line 90: private class Button Handler implements Event Handler Left angled bracket Action Event Right angled bracket. Line 91: left brace. Line 92: at symbol Override. Line 93: public void handle left parentheses Action Event event right parentheses. Line 94: left brace. Line 95: Grid Button button equals left parentheses Grid Button right parentheses event dot get Source left parentheses right parentheses semicolon. Line 96: if left parentheses game dot try To Play left parentheses button dot get Row left parentheses right parentheses comma button dot get Column left parentheses right parentheses right parentheses right parentheses. Line 97: update left parentheses right parentheses semicolon. Line 98: right brace. Line 99: right brace. Line 100: right brace.

Back to Example

The method is in the java f x dot scene dot layout package. The table has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Return value: Observable List, left angled bracket, Node, right angled bracket. Method name and argument list: get Children, left parentheses, right parentheses, returns the children of this pane as an Observable List of Nodes.

Back to Table

The methods are in the java f x dot scene dot control package. The constructor is as follows: Alert, left parentheses, Alert dot Alert Type type, right parentheses, constructs an Alert of type type. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: set Title, left parentheses, String title, right parentheses, sets the title of this dialog box. Row 2. Return value: void. Method name and argument list: set Header Text, left parentheses, String message, right parentheses, sets the text to display in the header of this dialog box. If message is the empty String, no space is allocated for the header and the dialog box will be smaller. Row 3. Return value: void. Method name and argument list: set Content Text, left parentheses, String message, right parentheses, sets the text to display inside dialog box. Row 4. Return value: Optional, left angled bracket, uppercase r, right angled bracket. Method name and argument list: show And Wait, left parentheses, right parentheses, shows this dialog box and waits for the user's response.

Back to Table

Program code. The code contains the following keywords: import, public, class, extends, void, new, static, String. Line 1: Forward slash. Asterisk. Play Tile Puzzle class. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: blank. Line 5: import java f x dot application dot Application, semicolon. Line 6: import java f x dot scene dot Scene, semicolon. Line 7: import java f x dot stage dot Stage, semicolon. Line 8: blank. Line 9: public class Play Tile Puzzle extends Application. Line 10: left brace. Line 11: blank. Line 12, indented once: at symbol, Override. Line 13, indented once: public void start, left parentheses, Stage stage, right parentheses. Line 14, indented once: left brace. Line 15, indented twice: Tile Puzzle puzzle, equals, new, Tile Puzzle, left parentheses, 3, right parentheses, semicolon. Line 16, indented twice: Tile Puzzle View Controller root. Line 17, indented thrice: equals, new Tile Puzzle View Controller, left parentheses, puzzle, right parentheses, semicolon. Line 18: blank. Line 19, indented twice: Scene scene, equals, new, Scene, left parentheses, root, comma, 350, comma, 350, right parentheses, semicolon. Line 20, indented twice: stage dot set Title, left parentheses, double quotes, The tile puzzle game, double quotes, right parentheses, semicolon. Line 21, indented twice: stage dot set Scene, left parentheses, scene, right parentheses, semicolon. Line 22, indented twice: stage dot show, left parentheses, right parentheses, semicolon. Line 23, indented once: right brace. Line 24: blank. Line 25, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 26, indented once: left brace. Line 27, indented twice: launch, left parentheses, a r g s, right parentheses, semicolon. Line 28, indented once: right brace. Line 29: right brace.

Back to Example

Program code. The code contains the following keywords: import, public, class, extends, private, static, final, int, double, super, this, return, double, void. Line 1: Forward slash. Asterisk. Sprite class. Line 2: Anderson, Franceschi. Line 3: Forward slash. Asterisk. Line 4: import java f x dot scene dot canvas dot asterisk, semicolon. Line 5: import java f x dot scene dot paint dot asterisk, semicolon. Line 6: blank. Line 7: public class Sprite extends Canvas. Line 8: left brace. Line 9: blank. Line 10, indented once: private static final int WIDTH, equals, 95, comma, HEIGHT, equals, 138, semicolon. Line 11, indented once: private int s X, semicolon. Line 12, indented once: private int s Y, semicolon. Line 13, indented once: private double scale, semicolon. Line 14: blank. Line 15, indented once: Forward slash. 2 asterisks. default constructor. Line 16, indented once: Asterisk. s X, equals, s Y, equals, 0, semicolon. scale is set to 1. Line 17, indented once: Asterisk. Forward slash. Line 18, indented once: public Sprite, left parentheses, right parentheses. Line 19, indented once: left brace. Line 20, indented twice: super, left parentheses, WIDTH, comma, HEIGHT, right parentheses, semicolon. Line 21, indented twice: scale, equals, 1, semicolon. Line 22, indented once: right brace. Line 23: blank. Line 24, indented once: Forward slash. Asterisk. overloaded constructor. Line 25, indented once: Asterisk. accepts values for starting x and y coordinates. Line 26, indented once: Asterisk. and scale. Line 27, indented once: Asterisk. Forward slash. Line 28, indented once: public Sprite, left parentheses, int s X, comma, int s Y, comma, double scale, right parentheses. Line 29, indented once: left brace. Line 30, indented twice: super, left parentheses, right parentheses, semicolon. Line 31, indented twice: set Coordinates, left parentheses, s X, comma, s Y, right parentheses, semicolon. Line 32, indented twice: this dot scale, equals, left parentheses, scale greater than 0.0, question mark, scale, colon, 1, right parentheses, semicolon. Line 33, indented twice: set Width, left parentheses, this dot scale asterisk WIDTH, right parentheses, semicolon. Line 34, indented twice: set Height, left parentheses, this dot scale asterisk HEIGHT, right parentheses, semicolon. Line 35, indented once: right brace. Line 36: blank. Line 37, indented once: Forward slash. Asterisk. set Coordinates. Line 38, indented once: Asterisk. accepts new values for starting x and y, semicolon. Line 39, indented once: Asterisk. returns a reference to this object. Line 40, indented once: Asterisk. Forward slash. Line 41, indented once: public Sprite set Coordinates, left parentheses, int s X, comma, int s Y, right parentheses. Line 42, indented once: left brace. Line 43, indented twice: this dot s X, equals, s X, semicolon. Line 44, indented twice: this dot s

Y, equals, s Y, semicolon. Line 45, indented twice: return this, semicolon. Line 46, indented once: right brace. Line 47: blank. Line 48, indented once: Forward slash. Asterisk. mutator for scale. Line 49, indented once: Asterisk. returns a reference to this object. Line 50, indented once: Asterisk. Forward slash. Line 51, indented once: public Sprite set Scale, left parentheses, double scale, right parentheses. Line 52, indented once: left brace. Line 53, indented twice: this dot scale, equals, left parentheses, scale greater than 0.0, question mark, scale, colon, this dot scale, right parentheses, semicolon. Line 54, indented twice: return this, semicolon. Line 55, indented once: right brace. Line 56: blank. Line 57, indented once: Forward slash. Asterisk. draw method. Line 58, indented once: Asterisk. draws Sprite at current s X and s Y. Line 59, indented once: Asterisk. multiplying lengths by scale. Line 60, indented once: Asterisk. accepts the Graphics Context of the canvas. Line 61, indented once: Asterisk. Forward slash. Line 62, indented once: public void draw, left parentheses, Graphics Context g c, right parentheses. Line 63, indented once: left brace. Line 64, indented twice: g c dot set Fill, left parentheses, Color dot CORAL, right parentheses, semicolon. 2 forward slashes, body. Line 65, indented twice: g c dot fill Oval, left parentheses, s X, comma, s Y plus 15 asterisk scale, comma, 90 asterisk scale, comma, 120 asterisk scale, right parentheses, semicolon. Line 66, indented twice: g c dot set Fill, left parentheses, Color dot DARK GOLDEN ROD, right parentheses, semicolon. 2

forward slashes, hat. Line 67, indented twice: g c dot fill Rect, left parentheses, s X plus 23 asterisk scale, comma, s Y, comma, 45 asterisk scale, comma, 22, asterisk, scale, right parentheses, semicolon. Line 68, indented twice: g c dot set Stroke, left parentheses, Color dot DARK GOLDEN ROD, right parentheses, semicolon. 2 forward slashes, hat brim. Line 69, indented twice: g c dot set Line Width, left parentheses, 3, right parentheses, semicolon. Line 70, indented twice: g c dot stroke Line, left parentheses, s X, comma, s Y plus 23 asterisk scale, comma. Line 71, indented multiple times: s X plus 90 asterisk scale, comma, s Y plus 23 asterisk scale, right parentheses, semicolon. Line 72, indented twice: g c dot set Fill, left parentheses, Color dot CHOCOLATE, right parentheses, semicolon. 2 forward slashes, eye. Line 73, indented twice: g c dot fill Oval, left parentheses, s X plus 60 asterisk scale, comma, s Y plus 45 asterisk scale, comma. Line 74, indented multiple times: 18 asterisk scale, comma, 12 asterisk scale, right parentheses, semicolon. Line 74, indented twice: g c dot set Fill, left parentheses, Color dot DARK SALMON, right parentheses, semicolon, 2 forward slashes, feet. Line 76, indented twice: g c dot set Line Width, left parentheses, 1, right parentheses, semicolon. Line 77, indented twice: g c dot fill Oval, left parentheses, s X plus 45 asterisk scale, comma, s Y plus 125 asterisk scale, comma. Line 78, indented multiple times: 45 asterisk scale, comma, 12 asterisk scale, right parentheses, semicolon. Line 77, indented twice: g c dot fill Oval, left parentheses, s X plus

45 asterisk scale, comma, s Y plus 125 asterisk scale, comma. Line 78, indented multiple times: 45 asterisk scale, comma, 12 asterisk scale, right parentheses, semicolon. Line 79, indented twice: g c dot stroke Oval, left parentheses, s X plus 45 asterisk scale, comma, s Y plus 125 asterisk scale, comma. Line 80, indented multiple times: 45 asterisk scale, comma, 12 asterisk scale, right parentheses, semicolon. Line 81, indented twice: g c dot fill Oval, left parentheses, s X plus 27 asterisk scale, comma, s Y plus 127 asterisk scale, comma. Line 82, indented multiple times: 45 asterisk scale, comma, 12 asterisk scale, right parentheses, semicolon. Line 83, indented twice: g c dot stroke Oval, left parentheses, s X plus 27 asterisk scale, comma, s Y plus 127 asterisk scale, comma. Line 84, indented multiple times: 45 asterisk scale, comma, 12 asterisk scale, right parentheses, semicolon. Line 85, indented once: right brace. Line 86: right brace.

Back to Example

Line 1: left angled bracket, question mark, x m l version, equals, double quotes, 1.0, double quotes, encoding, equals, double quotes, U T F hyphen 8, double quotes, question mark, right angled bracket. Line 2: left angled bracket, question mark, import java f x dot scene dot control dot asterisk question mark, right angled bracket. Line 3: left angled bracket, question mark, import java f x dot scene dot layout dot asterisk question mark, right angled bracket. Line 4: blank. Line 5: left angled bracket, Border Pane x m l n s, colon, f x, equals, double quotes, h t t p, colon, 2 forward slashes, java f x dot com, forward slash, f x m l, double quotes. Line 6, indented once: f x, colon, controller, equals, double quotes, Animation Controller, double quotes, right angled bracket. Line 7: left angled bracket, center, right angled bracket. Line 8, indented once: left angled bracket, V Box f x, colon, i d, equals, double quotes, canvas Container, double quotes, alignment, equals, double quotes, center, double quotes. Line 9, indented twice: style, equals, double quotes, hyphen, f x hyphen border hyphen color, colon, navy, semicolon, hyphen, f x hyphen, border hyphen width, colon, 1, double quotes, forward slash, right angled bracket. Line 10: left angled bracket, forward slash, center, right angled bracket. Line 11: left angled bracket, left, right angled bracket. Line 12, indented once: left angled bracket, V Box spacing, equals, double quotes, 10, double quotes, alignment, equals, double quotes, center, double quotes. Line 13, indented twice: style, equals, double quotes, hyphen, f x, hyphen, border, hyphen, color, colon, navy, semicolon, hyphen, f x, hyphen, border, hyphen, width, colon, 1, semicolon, hyphen, f x, hyphen, padding, colon, 8, double quotes, right angled bracket. Line 14: blank. Line 15, indented once: left angled bracket, Button text, equals, double quotes, Fade, double quotes, f x, colon, i d equals, double quotes, fade Button, double quotes, forward slash, right angled bracket. Line 16, indented once: left angled bracket, Button text, equals, double quotes, Rotate, double quotes, f x, colon, i d equals, double quotes, rotate Button, double quotes, forward slash, right angled bracket. Line 17, indented once: left angled bracket, Button text, equals, double quotes, Scale, double quotes, f x, colon, i d equals, double quotes, scale Button, double quotes, forward slash, right angled bracket. Line 18, indented once: left angled bracket, Button text, equals, double quotes, Combine Animations, double quotes, f x, colon, i d equals, double quotes, combine Button, double quotes, forward slash, right angled bracket. Line 19, indented once: left angled bracket, Button text, equals, double quotes, Start Translate, double quotes, f x, colon, i d equals, double quotes, start Translate Button, double quotes, forward slash, right angled bracket. Line 20, indented once: left angled bracket, Button text, equals, double quotes, Stop Translate, double quotes, f x, colon, i d equals, double quotes, stop Translate Button, double quotes, forward slash, right angled bracket. Line 21, indented twice: left angled bracket, forward slash, V Box, right angled bracket. Line 22, indented once: left angled bracket, forward slash, left, right angled bracket. Line 23: left angled bracket, forward slash, Border Pane, right angled bracket.

Back to Example

The methods are in the java f x dot animation package. The table listing the comon methods of animation subclasses has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: play, left parentheses, right parentheses, plays the animation from its current position. Row 2. Return value: void. Method name and argument list: set Auto Reverse, left parentheses, boolean value, right parentheses, if value is true, animation reverses on alternate cycles; if false, animation repeats from start position on each cycle. Row 3. Return value: void. Method name and argument list: set Cycle Count, left parentheses, int value, right parentheses, sets the animation to execute value times; if set to Animation. INDEFINITE, animation will run until stopped. Row 4. Return value: void. Method name and argument list: set On Finished, left parentheses, Event Handler, left angled bracket, Action Event, right angled bracket, handler, right parentheses, registers event handler's code to run when the animation finishes. Row 5. Return value: void. Method name and argument list: stop, left parentheses, right parentheses, stops the animation, if playing.

Back to Table

Program code. The code contains the following keywords: import, public, class, private, void, new, true, false, boolean. Line 1: Forward slash. Asterisk. Animation Controller. Line 2: Anderson, Franceschi. Line 3: Asterisk. Forward slash. Line 4: import java dot net dot U R L, semicolon. Line 5: blank. Line 6: import java f x dot animation dot asterisk, semicolon. Line 7: import java f x dot event dot asterisk, semicolon. Line 8: import java f x dot f x m l dot asterisk, semicolon. Line 9: import java f x dot scene dot canvas dot asterisk, semicolon. Line 10: import java f x dot scene dot control dot asterisk, semicolon. Line 11: import java f x dot scene dot layout dot asterisk, semicolon. Line 12: import java f x dot scene dot media dot Audio Clip, semicolon. Line 13: import java f x dot util dot Duration, semicolon. Line 14: blank. Line 15: public class Animation Controller. Line 16: left brace. Line 17, indented once: at symbol, F X M L private Button fade Button, comma, rotate Button, comma. Line 18, indented multiple times: scale Button, comma, combine Button, comma. Line 19, indented multiple times: start Translate Button, comma, stop Translate Button, semicolon. Line 20, indented once: at symbol, F X M L private V Box canvas Container, semicolon. Line 21: blank. Line 22, indented once: private Fade Transition f t Fade, semicolon. Line 23, indented once: private Rotate Transition r t Half, comma, r t Full, semicolon. Line 24, indented once: private Translate Transition t t Straight, comma, t t Diagonal, semicolon. Line 25, indented once: private Scale Transition s t Larger, comma, s t Smaller, comma, s t Squeeze, semicolon. Line 26, indented once: private sequential Transition sequential, semicolon. Line 27, indented once: private Parallel Transition parallel, semicolon. Line 28: blank. Line 29, indented once: private Audio Clip sound, semicolon. Line 30: blank. Line 31, indented once: at symbol, F X M L public void initialize, left parentheses, right parentheses. Line 32, indented once: left brace. Line 33, indented twice: 2 forward slashes, Sprite extends Canvas. Line 34, indented twice: Sprite sprite, equals, new Sprite, left parentheses, 0, comma, 0, comma, 1, right parentheses, semicolon. Line 35, indented twice: Graphics Context g c, equals, sprite dot get Graphics Context 2 D, left parentheses, right parentheses, semicolon. Line 36, indented twice: sprite dot draw, left parentheses, g c, right parentheses, semicolon. Line 37, indented twice: canvas Container dot get Children, left parentheses, right parentheses, dot, add, left parentheses, sprite, right parentheses, semicolon. Line 38: blank. Line 39, indented twice: 2 forward slashes, fade for 3 seconds, comma, then reverse. Line 40, indented twice: f t Fade, equals, new Fade Transition, left parentheses, Duration dot seconds, left parentheses, 3, right parentheses, comma, sprite, right parentheses, semicolon. Line 41, indented twice: f t Fade dot set From Value, left parentheses, 1.0, right parentheses, semicolon. Line 42, indented twice: f t Fade dot set To Value, left parentheses, 0.1, right parentheses, semicolon. Line 43, indented twice: f t Fade dot set Cycle Count, left parentheses, 2, right parentheses, semicolon. Line 44, indented twice: f t Fade dot set Auto Reverse, left parentheses, true, right parentheses, semicolon. Line 45, indented twice: fade Button dot set On Action, left parentheses, event, hyphen, right angled bracket, run Transition, left parentheses, f t Fade, right parentheses, right parentheses, semicolon. Line 46: blank. Line 47, indented twice: 2 forward slashes, sound effect. Line 48, indented twice: U R L resource, equals, get Class, left parentheses, right parentheses, dot, get Resource, left parentheses, double quotes, whoosh dot m 4 a, double quotes, right parentheses, semicolon. Line 49, indented twice: sound, equals, new Audio Clip, left parentheses, resource dot to String, left parentheses, right parentheses, right parentheses, semicolon. Line 50, indented twice: sound dot set Cycle Count, left parentheses, 2, right parentheses, semicolon. Line 51: blank. Line 52, indented twice: 2 forward slashes, rotate 180 degrees and back. Line 53, indented twice: r t Half, equals, new Rotate Transition, left parentheses, Duration dot millis, left parentheses, 3000, right parentheses, comma, sprite, right parentheses, semicolon. Line 54, indented twice: r t Half dot set From Angle, left parentheses, 0, right parentheses, semicolon. Line 55, indented twice: r t Half dot set To Angle, left parenthess, 180, right parentheses, semicolon. Line 56, indented twice: r t Half dot set Cycle Count, left parentheses, 2, right parentheses, semicolon. Line 57, indented twice: r t Half dot set Auto Reverse, left parentheses, true, right parentheses, semicolon. Line 58, indented twice: rotate Button dot set On Action, left parentheses, event, hyphen, right angled bracket. Line 59, indented multiple times: left brace. Line 60, indented multiple times: run Transition, left parentheses, r t Half, right parentheses, semicolon. Line 61, indented multiple times: sound dot play, left parentheses, right parentheses, semicolon. Line 62, indented multiple times: right brace. Line 63, indented multiple times: right parentheses, semicolon. Line 64:

blank. Line 65, indented twice: 2 forward slashes, reduce size to 1 forward slash 2, return to start size. Line 66, indented twice: s t Smaller, equals, new Scale Tansition, left parentheses, Duration dot millis, left parentheses, 1500, right parentheses, comma, sprite, right parentheses, semicolon. Line 67, indented twice: s t Smaller dot set T o X, left parentheses, 0.5 f, right parentheses, semicolon. Line 68, indented twice: s t Smaller dot set T o Y, left parentheses, 0.5 f, right parentheses, semicolon. Line 69, indented twice: s t Smaller dot set Cycle Count, left parentheses, 2, right parentheses, semicolon. Line 70, indented twice: s t Smaller dot set Auto Reverse, left parentheses, true, right parentheses, semicolon. Line 71: blank. Line 72, indented twice: 2 forward slashes, reduce x size to 1 forward slash 2, return to start size. Line 73, indented twice: s t Squeeze, equals, new Scale Transition, left parentheses, Duration dot millis, left parentheses, 1500, right parentheses, comma, sprite, right parentheses, semicolon. Line 74, indented twice: s t Squeeze dot set T o X, left parentheses, 0.5 f, right parentheses, semicolon. Line 75, indented twice: s t Squeeze dot set Cycle Count, comma, 2, right parentheses, semicolon. Line 76, indented twice: s t Squeeze dot set Auto Reverse, left parentheses, true, right parentheses, semicolon. Line 77: blank. Line 78, indented twice: sequential, equals, new sequential Transition, left parentheses, s t Smaller, comma, s t Squeeze, right parentheses, semicolon. Line 79, indented twice: scale Button dot set On Action, left parentheses, event, hyphen, right angled bracket, run Transition, left parentheses, sequential, right parentheses, right parentheses, semicolon. Line 80: blank. Line 81, indented twice: 2 forward slashes, move right 80 pixels, then return to start. Line 82, indented twice: t t Straight, equals, new Translate Transition, left parentheses, Duration dot seconds, left parentheses, 3, right parentheses, comma, sprite, right parentheses, semicolon. Line 83, indented twice: t t Straight dot set From X, left parentheses, 0, right parentheses, semicolon. Line 84, indented twice: t t Straight dot set T o X, left parentheses, 80, right parentheses, semicolon. Line 85, indented twice: t t Straight dot set Cycle Count, left parentheses, 2, right parentheses, semicolon. Line 86, indented twice: t t Straight dot set Auto Reverse, left parentheses, true, right parentheses, semicolon. Line 87: blank. Line 88, indented twice: 2 forward slashes, rotate 360 degrees. Line 89, indented twice: r t Full, equals, new Rotate Transition, left parentheses, Duration dot millis, left parentheses, 3000, right parentheses, comma, sprite, right parentheses, semicolon. Line 90, indented twice: r t Full dot set By Angle, left parentheses, 360, right parentheses, semicolon. Line 91: blank. Line 92, indented twice: 2 forward slashes, increase size by half, return to start size. Line 93, indented twice: s t Larger, equals, new Scale Transition, left parentheses, Duration dot millis, left parentheses, 1500, right parentheses, comma, sprite, right parentheses, semicolon. Line 94, indented twice: s t Larger dot set T o X, left parentheses, 1.5 f, right parentheses, semicolon. Line 95, indented twice: s t Larger dot set T o Y, left parentheses, 1.5 f, right parentheses, semicolon. Line 96, indented twice: s t Larger dot set Cycle Count, left parentheses, 2, right parentheses, semicolon. Line 97, indented twice: s t Larger dot set Auto Reverse, left parentheses, true, right parentheses, semicolon. Line 98: blank. Line 99, indented twice: parallel equals new Parallel Transition, left parentheses, t t Straight, comma, r t Full, comma, s t Larger, right parentheses, semicolon. Line 100, indented twice: combine Button dot set On Action, left parentheses, event, hyphen, right angled bracket, run Transition, left parentheses, parallel, right parentheses, right parentheses, semicolon. Line 101: blank. Line 102, indented twice: 2 forward slashes, move diagonally up 100 pixels, then return to start. Line 103, indented twice: t t Diagonal, equals, new Translate Transition, left parentheses, Duration dot millis, left parentheses, 1500, right parentheses, comma, sprite, right parentheses, semicolon. Line 104, indented twice: t t Diagonal dot set From X, left parentheses, 0, right parentheses, semicolon. Line 105, indented twice: t t Diagonal dot set T o X, left parentheses, 100, right parentheses, semicolon. Line 106, indented twice: t t Diagonal dot set From Y, left parentheses, 0, right parentheses, semicolon. Line 107, indented twice: t t Diagonal dot set To Y, left parentheses, negative 100, right parentheses, semicolon. Line 108, indented twice: t t Diagonal dot set Cycle Count, left parentheses, Animation dot INDEFINITE, right parentheses, semicolon. Line 109, indented twice: t t Diagonal dot set Auto Reverse, left parentheses, true, right parentheses, semicolon. Line 110, indented twice: stop Translate Button dot set Disable, left parentheses, true, right parentheses, semicolon. Line 111, indented twice: start Translate Button dot set On Action, left parentheses, event, hyphen, right angled bracket. Line 112, indented multiple times: left brace. Line 113, indented multiple times: stop Translate Button dot set Disable, left parentheses, false, right parentheses, semicolon. Line 114, indented multiple times: run Transition, left parentheses, t t Diagonal, right parentheses, semicolon. Line 115, indented multiple times: right brace, right parentheses, semicolon. Line 116: blank. Line 117, indented twice: stop Translate Button dot set On Action, left parentheses, event, hyphen, right angled bracket. Line 118, indented multiple times: left brace. Line 119, indented multiple times: stop Translate Button dot set Disable, left parentheses, true, right parentheses, semicolon. Line 120, indented multiple times: t t Diagonal dot stop, left parentheses, right parentheses, semicolon. Line 121, indented multiple times: disable Buttons, left parentheses, false, right parentheses, semicolon. Line 122, indented multiple times: right brace, right parentheses, semicolon. Line 123, indented twice: right brace. Line 124: blank. Line 125, indented twice: Forward slash. 2 asterisks. Line 126, indented once: Asterisk. run Transition. Line 127, indented once: Asterisk. at symbol, param t, comma, the transition to run. Line 128, indented once: Asterisk. disables buttons, sets animation finished event handler. Line 129, indented once: Asterisk. plays the animation. Line 130, indented once: Asterisk. Forward slash. Line 131, indented once: public void run Transition, left parentheses, Transition t, right parentheses. Line 132, indented once: left brace. Line 133, indented twice: disable Buttons, left parentheses, true, right parentheses, semicolon. Line 134, indented twice: t dot set On Finished, left parentheses, event, hyphen, right angled bracket, disable Buttons, left parentheses, false, right parentheses, right parentheses, semicolon. Line 135, indented twice: t dot play, left parentheses, right parentheses, semicolon. Line 136, indented once: right brace. Line 137: blank. Line 138, indented once: Forward slash. 2 asterisks. disable Buttons. Line 139, indented once: Asterisk. at symbol, param mode true to disable, false to enable. Line 140, indented once: Asterisk. Forward slash. Line 141, indented once: public void disable Buttons, left parentheses, boolean mode, right parentheses. Line 142, indented once: left brace. Line 143, indented twice: fade Button dot set Disable, left parentheses, mode, right parentheses, semicolon. Line 144, indented twice: start Translate Button dot set Disable, left parentheses, mode, right parentheses, semicolon. Line 145, indented thrice: rotate Button dot set Disable, left parentheses, mode, right parentheses, semicolon. Line 146, indented thrice: scale Button dot set Disable, left parentheses, mode, right parentheses, semicolon. Line 147, indented thrice: combine Button dot set Disable, left parentheses, mode, right parentheses, semicolon. Line 148, indented once: right brace. Line 149: right brace.

Back to Example

The methods are in the java f x dot animation package. The constructor of the Fade Transition class is as follows: Fade Transition, left parentheses, Duration time, comma, Node node, right parentheses, creates a Fade Transition that will animate node for a duration of time. The table listing the selected methods of the Fade Transition class has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: set From Value, left parentheses, double opacity, right parentheses, sets the beginning opacity of the animation; the opacity can range from 0.1, fully opaque, to 0.0, fully transparent. Row 2. Return value: void. Method name and argument list: set To Value, left parentheses, double opacity, right parentheses, sets the ending opacity of the animation; the opacity can range from 1.0, fully opaque, to 0.0, fully transparent.

Back to Table

The methods are in the java dot util package. The table has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1.

Return value: Duration. Method name and argument list: millis, left parentheses, double m s, right parentheses, creates a Duration object for m s milliseconds. Row 2. Return value: Duration. Method name and argument list: seconds, left parentheses, double s e c s, right parentheses, creates a Duration object for s e c s seconds. Row 3. Return value: Duration. Method name and argument list: minutes, left parentheses, double m i n s, right parentheses, creates a Duration object for m i n s minutes. Row 4. Return value: Duration. Method name and argument list: hours, left parentheses, double h r s, right parentheses, creates a Duration object for h r s hours.

Back to Table

The method is in the java f x dot event package. The method has 2 columns labeled Return value and Method name and argument list. The row entry is as follows. Return value: void. Method name and argument list: handle, left parentheses, uppercase t event, right parentheses, provides code to run when event occurs on a registered control.

Back to Table

The methods are in the java f x dot scene dot media package. The Audio Clip constructor is as follows: Audio Clip, left parentheses, String source, right parentheses, loads an audio clip from the U R L represented by source.

The table listing the selected methods of the Audio Clip Class has 2 columns labeled Return value and Method name and argument list. The row entries are as follows. Row 1. Return value: void. Method name and argument list: set Cycle Count, left parentheses, int count, right parentheses, count is the number of times to play the sound. Row 2. Return value: void. Method name and argument list: play, left parentheses, right parentheses, plays the sound. Row 3. Return value: void. Method name and argument list: stop, left parentheses, right parentheses, stops the sound, if playing.

Back to Table

A table detailing a constructor and methods of the Rotate Transition class. Rotate Transition class belongs to the java f x dot animation package. Constructor. Rotate Transition, left parenthesis, Duration, time, comma, Node, node, right parenthesis, creates a Rotate Transition that will animate node for a duration of time. A table with selected methods of the Rotate Transition class. The table has 3 rows and 2 columns. It has the following column headings. Return value. Method name and argument list. The row details in the table are the following. Row 1: Return value, void; Method name and argument list, set From Angle, left parenthesis, double, start Angle, right parenthesis, sets the angle from which to begin the rotation. Row 2: Return value, void; Method name and argument list, set To Angle, left parenthesis, double, end Angle, right parenthesis, sets the angle at which to end the rotation. Row 3: Return value, void; Method name and argument list, set By Angle, left parenthesis, double, angle, right parenthesis, sets the angle relative to the start at which to stop.

Back to Table

A table detailing a constructor and method of the Scale Transition class. Scale Transition class belongs to the java f x dot animation package. Constructor. Scale Transition, left parenthesis, Duration, time, comma, Node, node, right parenthesis, creates a Scale Transition that will animate node for a duration of time. A table with selected methods of the Scale Transition class. The table has 2 rows and 2 columns. It has the following column headings. Return value. Method name and argument list. The row details are the following. Row 1: Return value, void; Method name and argument list, set To X, left parenthesis, double, value, right parenthesis, sets the x scale at which to stop. Row 2: Return value, void; Method name and argument list, set To Y, left parenthesis, double, value, right parenthesis, sets the y scale at which to stop.

Back to Table

A table detailing the constructor of the Sequential Transition class. Sequential Transition belongs to the java f x dot animation package. Constructor. Sequential Transition, left parenthesis, Animation, dot, dot, dot, children, right parenthesis, creates a Sequential Transition that will play all children animations in sequence.

Back to Table

A table detailing a constructor and method of the Translate Transition class. Translate Transition belongs to the java x dot animation package. Constructor. Translate Transition, left parenthesis, Duration, time, comma, Node node, right parenthesis, creates a Translate Transition that will animate node for a duration of time. A table detailing selected methods of the Translate Transition class. The table has 4 rows and 2 columns. It has the following column headings. Return value. Method name and argument list. The row details in the table are the following. Row 1: Return value, void; Method name and argument list, set From X, left parenthesis, double, value, right parenthesis, sets the starting x value. Row 2: Return value, void; Method name and argument list, set To X, left parenthesis, double, value, right parenthesis, sets the ending x value. Row 3: Return value, void; Method name and argument list, set From Y, left parenthesis, double, value, right parenthesis, sets the starting y value. Row 4: Return value, void; Method name and argument list, set To Y, left parenthesis, double, value, right parenthesis, sets the ending y value.

Back to Table

A table detailing a constructor of the Parallel Transition class. Parallel Transition class belongs to the java f x dot animation package. Parallel Transition, left parenthesis, Animation, dot, dot, dot, children, right parenthesis, creates a Parallel Transition that will play all children animations simultaneously.

Back to Table

Code in the program Animation Sampler dot java. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Animation Sampler. Line 2: demonstrates various animations. Line 3: Anderson, comma, Franceschi. Line 4: Asterisk, forward slash. Line 5: Blank. Line 6: import, java, dot, net, dot, U R L, semicolon. Line 7: import, java f x, dot, application, dot, asterisk, semicolon. Line 8: import, java f x, dot, f x m l, dot, asterisk, semicolon. Line 9: import, java f x, dot, scene, dot, layout, dot, asterisk, semicolon. Line 10: import, java f x, dot, scene, dot, asterisk, semicolon. Line 11: import, java f x, dot, stage, dot, asterisk, semicolon. Line 12: Blank. Line 13: public, class, Animation Sampler, extends, Application. Line 14: Left brace. Line 15, indented once: at, Override. Line 16, indented once: public, void, start, left parenthesis, Stage, stage, right parenthesis, forward slash, forward slash, throws Exception. Line 17, indented once: Left brace. Line 18, indented twice: try. Line 19, indented twice: Left brace. Line 20, indented 3 times: U R L, u r l, equals. Line 21, indented 4 times: get Class, left parenthesis, right parenthesis, dot, get Resource, left parenthesis, open quotes, f x m l underscore animation dot f x m l, close quotes, right parenthesis, semicolon. Line 22, indented 3 times: Border Pane, root, equals, F X M L Loader, dot, load, left parenthesis, u r l, right parenthesis, semicolon. Line 23, indented 3 times: Scene, scene, equals, new, Scene, left parenthesis, root, comma, 800, comma, 600, right parenthesis, semicolon. Line 24, indented 3 times: stage, dot, set Title, left parenthesis, open quotes, Animation Sampler, close quotes, right parenthesis. Line 25, indented 3 times: stage, dot, set Scene, left parenthesis, scene, right parenthesis, semicolon. Line 26, indented 3 times: stage, dot, show, left parenthesis, right parenthesis, semicolon. Line 27, indented twice: right brace. Line 28, indented twice: catch, left parenthesis, Exception, e, right parenthesis. Line 29, indented twice: Left brace. Line 30, indented 3 times: e, dot, print Stack Trace, left parenthesis, right parenthesis, semicolon. Line 31, indented twice: right brace. Line 32, indented once: right brace. Line 33: Blank. Line 34, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 35, indented once: Left brace. Line 36, indented twice: launch, left parenthesis, a r g s, right parenthesis, semicolon. Line 37, indented once: right brace. Line 38: right brace.

Back to Example

Code in the interface program Color Grid Game. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Color Grid Game interface. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java f x, dot, scene, dot, paint, dot, Color, semicolon. Line 6: Blank. Line 7: public, interface, Color Grid Game. Line 8: Left brace. Line 9, indented twice: public, String, get Title, left parenthesis, right parenthesis, semicolon. Line 10, indented twice: public, i n t, get Size, left parenthesis, right parenthesis, semicolon. Line 11, indented twice: public, i n t, get Number Of Colors, left parenthesis, right parenthesis, semicolon. Line 12, indented twice: public, String, get Label, left parenthesis, i n t, index, right parenthesis. Line 13, indented twice: public, i n t, get Index, left parenthesis, String, label, right parenthesis, semicolon. Line 14, indented twice: public, boolean, is Correct, left parenthesis, i n t, index, right parenthesis, semicolon. Line 15, indented twice: public, String, get Grid Hex Color, left parenthesis, i n t, row, comma, i n t, c o l, right parenthesis, semicolon. Line 16: right brace.

Back to Example

Code in the Color Frequency Game class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Color Frequency Game class. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java, dot, util, dot, Random, semicolon. Line 6: import, java f x, dot, scene, dot, paint, dot, Color, semicolon. Line 7: Blank. Line 8: public, class, Color Frequency Game, implements, Color Grid Game. Line 9: Left brace. Line 10, indented once: private, static, char, left square brace, right square brace, hex Digits, equals, Line 11, indented once: Left brace, open single quotes, 0, close single quotes, comma, open single quotes, 1, close sing quotes, comma, open single quotes, 2, close single quotes, comma, open single quotes, 3, close single quotes, comma, open single quotes, 4, close single quotes, comma, open single quotes, 5, close single quotes, comma, open single quotes, 6, close single quotes, comma, open single quotes, 7, close single quotes, comma, open single quotes, 8, close single quotes, comma, open single quotes, 9, close single quotes, comma. Line 12, indented 3 times: open single quotes, A, close single quotes, comma, open single quotes, B, close single quotes, comma, open single quotes, C, close single quotes, comma, open single quotes, D, close single quotes, comma, open single quotes, E, close single quotes, comma, open single quotes, F, close single quotes, right brace, semicolon. Line 13: Blank. Line 14, indented once: private, final, i n t, DEFAULT underscore SIZE, equals, 8, semicolon. Line 15, indented once: private, Color, left square brace, right square brace, left square brace, right square brace, grid Colors, semicolon. Line 16, indented once: private, Color, left square brace, right square brace, coloring Colors, semicolon. Line 17, indented once: private, i n t, left square brace, right square brace, counts, semicolon, forward slash, forward slash, color frequencies in grid Colors. Line 18: Blank. Line 19, indented once: private, String, left square brace, right square brace, labels, semicolon. Line 20: Blank. Line 21, indented once: Forward slash, asterisk, asterisk, Constructor. Line 22, indented once: Asterisk, at, param, size number of rows and columns in grid Colors. Line 23, indented once: Asterisk, at, param, colors array of colors. Line 24, indented once: Asterisk, at, param, labels the starting labels. Line 25, indented once: Asterisk, forward slash. Line 26, indented once: public, Color Frequency Game, left parenthesis, i n t, size, comma, Color, left square brace, right square brace, colors, comma. Line 27, indented a few times: String, left square brace, right square brace, labels, right parenthesis. Line 28, indented once: Left brace. Line 29, indented twice: if, left parenthesis, size, left angle bracket, equals, 0, right parenthesis. Line 30, indented 3 times: size, equals, DEFAULT underscore SIZE, semicolon. Line 31, indented twice: grid Colors, equals, new, Color, left square brace, size, right square brace, left square brace, size, right square brace, semicolon. Line 32, indented twice: coloring Colors, equals, colors, semicolon. Line 33, indented twice: counts, equals, new, i n t, left square brace, coloring Colors, dot, length, right square brace, semicolon. Line 34, indented once: fill Grid With Colors, left parenthesis, right parenthesis, semicolon. Line 35, indented twice: this, dot, labels, equals, labels, semicolon. Line 36, indented once: right brace. Line 37: Blank. Line 38, indented once:

Forward slash, asterisk, asterisk, fill Grid With Colors method. Line 39, indented once: Asterisk, randomly fills grid Colors with colors from coloring Colors. Line 40, indented once: Asterisk, forward slash. Line 41, indented once: public, void, fill Grid With Colors, left parenthesis, right parenthesis. Line 42, indented once: Left brace. Line 43, indented twice: Random, random, equals, new, Random, left parenthesis, right parenthesis, semicolon. Line 44, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, grid Colors, dot, length, semicolon, i, plus, plus, right parenthesis. Line 45, indented twice: left brace. Line 46, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, grid Colors, left square brace, i, right square brace, dot, length, semicolon, j, plus, plus, right parenthesis. Line 47, indented 3 times: Left brace. Line 48, indented 4 times: i n t, color Index, equals, random, dot, next I n t, left parenthesis, coloring Colors, dot, length, right parenthesis, semicolon. Line 49, indented 4 times: grid Colors, left square brace, i, right square brace, left square brace, j, right square brace, equals, coloring Colors, left square brace, color Index, right square brace, semicolon. Line 50, indented 4 times: counts, left square brace, color Index, right square brace, plus, equals, 1, semicolon. Line 51, indented 3 times: right brace. Line 52, indented twice: right brace. Line 53, indented once: right brace. Line 54: Blank. Line 55, indented once: Forward slash, asterisk, asterisk, get Size method. Line 56, indented once: Asterisk, at, return, length of grid Colors. Line 57, indented once: Asterisk, forward slash. Line 58, indented once: public, i n t, get Size, left parenthesis, right parenthesis. Line 59, indented once: Left brace. Line 60, indented twice: return, grid Colors, dot, length, semicolon. Line 61, indented once: right brace. Line 62: Blank. Line 63, indented once: Forward slash, asterisk, asterisk, get Number Of Colors method. Line 64, indented once: Asterisk, at, return, length of coloring Colors. Line 65, indented once: Asterisk, forward slash. Line 66, indented once: public, i n t, get Number Of Colors, left parenthesis, right parenthesis. Line 67, indented once: Left brace. Line 68, indented twice: return, coloring Colors, dot, length, semicolon. Line 69, indented once: right brace. Line 70: Blank. Line 71, indented once: Forward slash, asterisk, asterisk, get Grid Hex Color method. Line 72, indented once: Asterisk, at, param, row, an i n t, the row index. Line 73, indented once: Asterisk, at, param c o l, an i n t, the column index. Line 74, indented once: Asterisk, at, return a String, the hex equivalent of grid Color, left square brace, row, right square brace, left square brace, c o l, right square brace. Line 75, indented once: Asterisk, forward slash. Line 76, indented once: public, String, get Grid Hex Color, left parenthesis, i n t, row, comma, i n t, c o l, right parenthesis. Line 77, indented once: Left brace. Line 78, indented twice: String, color Text, equals, open quotes, hash, close quotes. Line 79, indented 3 times: plus, to Hex, left parenthesis, left parenthesis, i n t, right parenthesis, left parenthesis, 255, asterisk, grid Colors, left square brace, row, right square brace, left square brace, c o l, right square brace, dot, get Red, left parenthesis, right parenthesis, right parenthesis, right parenthesis. Line 80, indented 3 times: plus, to Hex, left parenthesis, left parenthesis, i n t, right parenthesis, left parenthesis, 255, asterisk, grid Colors, left square brace, row, right square brace, left square brace, c o l, right square brace, dot, get Green, left parenthesis, right parenthesis, right parenthesis, right parenthesis. Line 81, indented 3 times: plus, to Hex, left parenthesis, left parenthesis, i n t, right parenthesis, left parenthesis, 255, asterisk, grid Colors, left square brace, row, right square brace, left square brace, c o l, right square brace, dot, get Blue, left parenthesis, right parenthesis, right parenthesis, right parenthesis, semicolon. Line 82, indented twice: return, color Text, semicolon. Line 83, indented once: right brace. Line 84: Blank. Line 85, indented once: forward slash, asterisk, asterisk, to Hex method. Line 86, indented once: Asterisk, at, return, a String, the Hex equivalent of color Intensity. Line 87, indented once: Asterisk, forward slash. Line 88, indented once: public, String, to Hex, left parenthesis, i n t, color Intensity, right parenthesis. Line 89, indented once: Left brace. Line 90, indented twice: if, left parenthesis, color Intensity, greater than, 255, right parenthesis. Line 91, indented 3 times: color Intensity, equals, 255, semicolon. Line 92, indented twice: else, if, left parenthesis, color Intensity, less than, 0, right parenthesis. Line 93, indented 3 times: color Intensity, equals, 0, semicolon. Line 94: Blank. Line 95, indented twice: i n t, first Digit, equals, color Intensity, forward slash, 16, semicolon. Line 96, indented twice: i n t, second Digit, equals, color Intensity, percentage, 16, semicolon. Line 97, indented twice: return, String, dot, value Of, left parenthesis, hex Digits, left square brace, first Digit, right square brace, right parenthesis, plus, hex Digits, left square brace, second Digit, right square brace, semicolon. Line 98, indented once: right brace. Line 99: Blank. Line 100, indented once: Forward slash, asterisk, asterisk, highest Count method. Line 101, indented once: Asterisk, at return, the highest color frequency in the grid. Line 102: Asterisk, forward slash. Line 103, indented once: public, i n t, highest Count, left parenthesis, right parenthesis. Line 104, indented once: Left brace. Line 105, indented twice: i n t, max, equals, counts, left square brace, 0, right square brace, semicolon. Line 106, indented twice: for, left square brace, i n t, i, equals, 1, semicolon, i, less than, counts, dot, length, semicolon, i, plus, plus, right parenthesis. Line 107, indented twice: Left brace. Line 108, indented 3 times: if, left parenthesis, counts, left square brace, i, right square brace, greater than, max, right parenthesis. Line 109, indented 4 times: max, equals, counts, left square brace, i, right square brace, semicolon. Line 110, indented twice: right brace. Line 111, indented twice: return, max, semicolon. Line 112, indented once: right brace. Line 113: Blank. Line 114, indented once: Forward slash, asterisk, asterisk, is Correct method. Line 115, indented once: Asterisk, @, param, index, the index in the coloring Colors array to check. Line 116, indented once: Asterisk, at, return true if the frequency of the color for index. Line 117, indented once: Asterisk, is the highest, false otherwise. Line 118, indented once: Asterisk, forward slash. Line 119, indented once: public, boolean, is Correct, left parenthesis, i n t, index, right parenthesis. Line 120, indented once: Left brace. Line 121, indented twice: return, counts, left square brace, index, right square brace, equals, equals, highest Count, left parenthesis, right parenthesis, semicolon. Line 122, indented once: right brace. Line 123: Blank. Line 124, indented once: forward slash, asterisk, asterisk, get Title method. Line 125, indented once: Asterisk, at, return, a String representing a title for this object. Line 126, indented once: Asterisk, forward slash. Line 127, indented once: public, String, get Title, left parenthesis, right parenthesis. Line 128, indented once: Left brace. Line 129, indented twice: return, open quotes, What is the most frequent color, question mark, close quotes, semicolon. Line 130, indented once: right brace. Line 131: Blank. Line 132, indented once: Forward slash, asterisk, asterisk, get Label method. Line 133, indented once: Asterisk, at, param, index, comma, the index of the Color name. Line 134, indented once: Asterisk, at, return the color name. Line 135, indented once: asterisk, forward slash. Line 136, indented once: public, String, get Label, left parenthesis, i n t, index, right parenthesis. Line 137, indented once: left brace. Line 138, indented twice: return, labels, left square brace, index, right brace, semicolon. Line 139, indented once: right brace. Line 140: Blank. Line 141, indented once: Forward slash, asterisk,

asterisk, get Index method. Line 142, indented once: Asterisk, at, param label, a color name in the labels array. Line 143, indented once: Asterisk, at, return index of the color name in the labels array. Line 144, indented once: Asterisk, forward slash. Line 145, indented once: public, i n t, get Index, left parenthesis, String, label, right parenthesis. Line 146, indented once: Left brace. Line 147, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, labels, dot, length, semicolon, i, plus, plus, right parenthesis. Line 148, indented twice: Left brace. Line 149, indented 3 times: if, left parenthesis, labels, left square brace, i, right square brace, dot, equals, left parenthesis, label, right parenthesis, right parenthesis. Line 150, indented 4 times: return, i, semicolon. Line 151, indented twice: right brace. Line 152, indented twice: return, minus 1, semicolon. Line 153, indented once: right brace. Line 154: right brace.

Back to Example

Code in the Bordered Label Grid View class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Bordered Label Grid View class. Line 2: Asterisk, Reusable generic layout using a Border Pane's. Line 3: Asterisk left and center positions. Line 4: Asterisk, The left V Box is made up of a vertical array of buttons. Line 5: Asterisk, The center position is made up of a grid of labels. Line 6: Asterisk, Accessors are provided so that a Controller can access. Line 7: Asterisk, the array of buttons and the 2-dim array (grid) of labels. Line 8: Asterisk, Anderson, comma, Franceschi. Line 9: Asterisk, forward slash. Line 10: Blank. Line 11: import, java f x, dot, event, dot, asterisk, semicolon. Line 12: import, java f x, dot, scene, dot, control, dot, asterisk, semicolon. Line 13: import, java f x, dot, scene, dot, layout, dot, asterisk, semicolon. Line 14: import, java f x, dot, scene, dot, paint, dot, Color, semicolon. Line 15: Blank. Line 16: public, class, Bordered Label Grid View, extends, Border Pane. Line 17: Left brace. Line 18, indented once: private, V Box, button Panel, semicolon, forward slash, forward slash, left, holds array of buttons. Line 19, indented once: private, Button, left square brace, right square brace, buttons, semicolon. Line 20, indented once: private, Grid Pane, grid, semicolon, forward slash, forward slash, center, holds grid of labels. Line 21, indented once: private, Label, left square brace, right square brace, left square brace, right square brace, labels, semicolon, forward slash, forward slash, grid of labels. Line 22: Blank. Line 23, indented once: forward slash, forward slash, number Of Buttons and grid Size must be greater than 0. Line 24, indented once: public, Bordered Label Grid View, left parenthesis, i n t, number Of Buttons, comma, i n t, grid Size, right parenthesis. Line 25, indented once: Left brace. Line 26, indented twice: super, left parenthesis, right parenthesis, semicolon. Line 27, indented twice: grid, equals, new, Grid Pane, left parenthesis, right parenthesis, semicolon. Line 28: Blank. Line 29, indented twice: forward slash, forward slash, set up grid as grid Size by grid Size. Line 30, indented twice: Row Constraints, row, equals, new, Row Constraints, left parenthesis, right parenthesis, semicolon. Line 31, indented twice: row, dot, set Percent Height, left parenthesis, 100.0, forward slash, grid Size, right parenthesis, semicolon. Line 32, indented twice: Column Constraints, col, equals, new, Column Constraints, left parenthesis, right parenthesis, semicolon. Line 33, indented twice: c o l, dot, set Percent Width, left parenthesis, 100.0, forward slash, grid Size, right parenthesis, semicolon. Line 34, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, grid Size, semicolon, i, plus, plus, right parenthesis. Line 35, indented 3 times: grid, dot, get Row Constraints, left parenthesis, right parenthesis, dot, add, left parenthesis, row, right parenthesis, semicolon. Line 36, indented twice: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, grid Size, semicolon, j, plus, plus, right parenthesis. Line 37, indented 3 times: grid, dot, get Column Constraints, left parenthesis, right parenthesis, dot, add All, left parenthesis, c o l, right parenthesis, semicolon. Line 38: Blank. Line 39, indented twice: labels, equals, new, Label, left square brace, grid Size, right square brace, left square brace, grid Size, right square brace, semicolon. Line 40, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, labels, dot, length, semicolon, i, plus, plus, right parenthesis. Line 41, indented twice: Left brace. Line 42, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, labels, left square brace, i, right square brace, dot, length, semicolon, j, plus, plus, right parenthesis. Line 43, indented 3 times: Left brace. Line 44, indented 4 times: labels, left square brace, i, right square brace, left square brace, j, right square brace, equals, new, Label, left parenthesis, right parenthesis, semicolon. Line 45, indented 4 times: forward slash, forward slash, make label fill up available width and height. Line 46, indented 4 times: labels, left square brace, i, right square brace, left square brace, j, right square brace, dot, set Max Width, left parenthesis, Double, dot, MAX underscore VALUE, right parenthesis, semicolon. Line 47, indented 4 times: labels, left square brace, i, right square brace, left square brace, j, right square brace, dot, set Max Height, left parenthesis, Double, dot, MAX underscore VALUE, right parenthesis, semicolon. Line 48, indented 4 times: grid, dot, add, left parenthesis, labels, left square brace, i, right square brace, left square brace, j, right square brace, comma, j, comma, i, right parenthesis, semicolon. Line 49, indented 3 times: right brace. Line 50, indented twice: right brace. Line 51: Blank. Line 52, indented twice: button Panel, equals, new, V Box, left parenthesis, right parenthesis, semicolon. Line 53: Blank. Line 54, indented twice: buttons, equals, new, Button, left square brace, number Of Buttons, right square brace, semicolon. Line 55, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, buttons, dot, length, semicolon, i, plus, plus, right brace. Line 56, indented twice: left brace. Line 57, indented 3 times: buttons, left square brace, i, right square brace, equals, new, Button, left parenthesis, right parenthesis, semicolon. Line 58, indented 3 times: forward slash, forward slash, make button fill up available width and height. Line 59, indented 3 times: buttons, left square brace, i, right square brace, dot, set Max Width, left parenthesis, Double, dot, MAX underscore VALUE, right parenthesis, semicolon. Line 60, indented 3 times: buttons, left square brace, i, right square brace, dot, set Max Height, left parenthesis, Double, dot, MAX underscore VALUE, right parenthesis, semicolon. Line 61, indented 3 times: V Box, dot, set V grow, left parenthesis, buttons, left square brace, i, right square brace, comma, Priority, dot, ALWAYS, right parenthesis, semicolon. Line 62, indented 3 times: button Panel, dot, get Children, left parenthesis, right parenthesis, dot, add, left parenthesis, buttons, left square brace, i, right square brace, right parenthesis, semicolon. Line 63, indented twice: right brace. Line 64: Blank. Line 65, indented twice: set Left, left parenthesis, button Panel, right parenthesis, semicolon. Line 66, indented twice: set Center, left parenthesis, grid, right parenthesis, semicolon. Line 67, indented once: right brace. Line 68: Blank. Line 69, indented once: public, void, set Button Text, left parenthesis, i n t, row, comma, String, text, right parenthesis. Line 70, indented once: Left brace. Line 71, indented twice: buttons, left brace, row, right brace, dot, set Text, left parenthesis, text, right parenthesis, semicolon. Line 72, indented once: right brace. Line 73: Blank. Line 74, indented once: public, void, set Button

Listener, left parenthesis, i n t, row, comma, Event Handler, left angle bracket, Action Event, right angle bracket, e h, right parenthesis. Line 75, indented once: Left brace. Line 76, indented once: buttons, left square brace, row, right square brace, dot, set On Action, left parenthesis, e h, right parenthesis, semicolon. Line 77, indented once: right brace. Line 78: Blank. Line 79, indented once: public, void, set Label Background, left parenthesis, i n t, row, comma, i n t, c o l, comma, String, hex Color, right parenthesis. Line 80, indented once: Left brace. Line 81, indented twice: labels, left square brace, row, right square brace, left square brace, c o l, right square brace, dot, set Style, left parenthesis, open quotes, hyphen f x hyphen background hyphen color, colon, close quotes, plus, hex Color, right parenthesis, semicolon. Line 82, indented once: right brace. Line 83: right brace.

Back to Example

A table detailing the set V grow method of the V Box class. The V Box class belongs to the java f x dot scene dot layout package. Return value: void. Method name and argument list: set V grow, left parenthesis, Node, child, comma, Priority, value, right parenthesis, static method that sets the vertical grow priority of child, assuming that child is contained in a V Box. If set, the V Box will allocate additional space for child if space is available.

Back to Table

Code in the Color Grid Game Controller class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Color Grid Game Controller class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java f x, dot, event, dot, asterisk, semicolon. Line 6: import, java f x, dot, scene, dot, control, dot, Alert, dot, Alert Type, semicolon. Line 7: import, java f x, dot, scene, dot, control, dot, asterisk, semicolon. Line 8: import, java f x, dot, scene, dot, layout, dot, asterisk, semicolon. Line 9: import, java f x, dot, scene, dot, paint, dot, Color, semicolon. Line 10: Blank. Line 11: public, class, Color Grid Game Controller, implements, Event Handler, left angle bracket, Action Event, right angle bracket. Line 12: Left brace. Line 13, indented once: private, Color Grid Game, model, semicolon. Line 14, indented once: private, Bordered Label Grid View, view, semicolon. Line 15: Blank. Line 16, indented once: public, Color Grid Game Controller, left parenthesis, Color Grid Game, model, comma. Line 17, indented a few times: Bordered Label Grid View, view, right parenthesis. Line 18, indented once: Left brace. Line 19, indented twice: this, dot, model, equals, model, semicolon. Line 20, indented twice: this, dot, view, equals, view, semicolon. Line 21, indented twice: set Up Game, left parenthesis, right parenthesis, semicolon. Line 22, indented once: right brace. Line 23: Blank. Line 24, indented once: public, void, set Up Game, left parenthesis, right parenthesis. Line 25, indented once: left brace. Line 26, indented twice: set Up Labels, left parenthesis, right parenthesis, semicolon. Line 27, indented once: set Up Buttons, left parenthesis, right parenthesis, semicolon. Line 28, indented once: right brace. Line 29: Blank. Line 30, indented once: public, void, set Up Labels, left parenthesis, right parenthesis. Line 31, indented once: Left brace. Line 32, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, model, dot, get Size, left parenthesis, right parenthesis, semicolon, i, plus, plus, right parenthesis. Line 33, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, model, dot, get Size, left parenthesis, right parenthesis, semicolon, j, plus, plus, right parenthesis. Line 34, indented 4 times: view, dot, set Label Background, left parenthesis, i, comma, j, comma, model, dot, get Grid Hex Color, left parenthesis, i, comma, j, right parenthesis, right parenthesis, semicolon. Line 35, indented once: right brace. Line 36: Blank. Line 37, indented once: public, void, set Up Buttons, left parenthesis, right parenthesis. Line 38, indented once: Left brace. Line 39, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, model, dot, get Number Of Colors, left parenthesis, right parenthesis, semicolon, i, plus, plus, right parenthesis. Line 40, indented twice: Left brace. Line 41, indented 3 times: view, dot, set Button Text, left parenthesis, i, comma, model, dot, get Label, left parenthesis, i, right parenthesis, right parenthesis, semicolon. Line 42, indented 3 times: view, dot, set Button Listener, left parenthesis, i, comma, this, right parenthesis, semicolon. Line 43, indented twice: right brace. Line 44, indented once: Line 45: Blank. Line 46, indented once: public, void, handle, left parenthesis, Action Event, event, right parenthesis. Line 47, indented once: Left brace. Line 48, indented twice: Button, button, equals, left parenthesis, Button, right parenthesis, event, dot, get Source, left parenthesis, right parenthesis, semicolon. Line 49, indented twice: i n t, index, equals, model, dot, get Index, left parenthesis, button, dot, get Text, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 50: Blank. Line 51, indented twice: if, left parenthesis, model, dot, is Correct, left parenthesis, index, right parenthesis, right parenthesis. Line 52, indented a few times: show Message Dialog, left parenthesis, open quotes, Congratulations, close quotes, comma, open quotes, You won, close quotes, right parenthesis, semicolon. Line 53, indented twice: else. Line 54, indented a few times: show Message Dialog, left parenthesis, open quotes, Sorry, close quotes, comma, open quotes, You lost, close quotes, right parenthesis, semicolon. Line 55, indented twice: forward slash, forward slash, disable buttons here. Line 56, indented once: right brace. Line 57: Blank. Line 58, indented once: public, void, show Message Dialog, left parenthesis, String, title, comma, String, message, right parenthesis. Line 59, indented once: Left brace. Line 60, indented twice: Alert, alert, equals, new, Alert, left parenthesis, Alert Type, dot, INFORMATION, right parenthesis, semicolon. Line 61, indented twice: alert, dot, set Title, left parenthesis, title, right parenthesis, semicolon. Line 62, indented once: alert, dot, set Header Text, left parenthesis, open quotes, close quotes, right parenthesis, semicolon. Line 63, indented once: alert, dot, set Header Text, left parenthesis, message, right parenthesis, semicolon. Line 64, indented twice: alert, dot, show And Wait, left parenthesis, right parenthesis, semicolon. Line 65, indented once: right brace. Line 66: right brace.

Back to Example

Code in the Play Color Count class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Play Color Count class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java f x, dot, application, dot, Application, semicolon. Line 6: import, java f x, dot, scene, dot, paint, dot, Color, semicolon. Line 7: import, java f x, dot, scene, dot, Scene, semicolon. Line 8: import, java f x, dot, stage, dot, Stage, semicolon. Line 9: Blank. Line 10: public, class, Play Color Count, extends, Application. Line 11: Left brace. Line 12: Blank. Line 13, indented once: at, Override. Line 14, indented once: public, void, start, left parenthesis, Stage, stage, right parenthesis. Line 15, indented once: Left brace. Line 16, indented twice: Color, left square brace, right square brace, colors, equals, left brace, Color, dot, RED, comma, Color, dot, GREEN, comma, Color, dot, BLUE, right brace, semicolon. Line 17, indented twice: String, left square brace, right square brace, labels, equals, left brace, open quotes, RED, close quotes, comma, open quotes, GREEN, close quotes, comma, open quotes, BLUE, close quotes, right brace, semicolon. Line 18: Blank. Line 19, indented twice: Color Frequency Game, game. Line 20, indented 3 times: equals, new, Color Frequency Game, left parenthesis, 8, comma, colors, comma, labels, right parenthesis, semicolon. Line 21: Blank. Line 22, indented twice: Bordered Label Grid View root. Line 23, indented 3 times: new, Bordered Label Grid View, left parenthesis, colors, dot, length, comma, game, dot, get Size, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 24: Blank. Line 25, indented once: Color Grid Game Controller, controller. Line 26, indented 3 times: equals, new, Color Grid Game Controller, left parenthesis, game, root, right parenthesis, semicolon. Line 27: Blank. Line 28, indented twice: Scene, scene, equals, new, Scene, left parenthesis, root, comma, 450, comma, 425, right parenthesis, semicolon. Line 29, indented twice: stage, dot, set Title, left parenthesis, game, dot, get Title, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 30, indented twice: stage, dot, set Scene, left parenthesis, scene, right parenthesis, semicolon. Line 31, indented twice: stage, dot, show, left parenthesis, right parenthesis, semicolon. Line 32, indented once: right brace. Line 33: Blank. Line 34, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 35, indented once: Left brace. Line 36, indented twice:

launch, left parenthesis, a r g s, right parenthesis, semicolon. Line 37, indented once: right brace. Line 38: right brace.

Back to Example

Table detailing useful classes and methods for mouse and touch events. The package is java f x dot scene dot input. A table detailing methods of the Mouse Event class. The table has 4 rows and 2 columns. It has the following column headings. Return value. Method name and argument list. Row 1: Return value, double; Method name and argument list, get Scene X, left parenthesis, right parenthesis, returns the x coordinate of this Mouse Event in the Scene where it occurs. Row 2: Return value, double; Method name and argument list, get Scene Y, left parenthesis, right parenthesis, returns the y coordinate of this Mouse Event in the Scene where it occurs. Row 3: Return value, double; Method name and argument list, get X, left parenthesis, right parenthesis, returns the x coordinate of the Mouse Event relative to the node on which the mouse listener is registered. Row 4: Return value, double; Method name and argument list, get Y, left parenthesis, right parenthesis, returns the y coordinate of the Mouse Event relative to the node on which the mouse listener is registered. Method of the Touch Event class. Return value: Touch Point. Method name and argument list: get Touch Point, left parenthesis, right parenthesis, returns the touch point for this Touch Event. Table detailing methods of the Touch Point class. The table has

4 rows and 4 columns. It has the following column headings. Return value. Method name and argument list. The row details in the table are the following. Row 1: Return value, double; Method name and argument list, get Scene X, left parenthesis, right parenthesis, returns the x coordinate of this Touch Point in the Scene in which it occurs. Row 2: Return value, double; Method name and argument list, get Scene Y, left parenthesis, right parenthesis, returns the y coordinate of this Touch Point in the Scene in which it occurs. Row 3: Return value, double; Method name and argument list, get X, left parenthesis, right parenthesis, returns the x coordinate of the Touch Point relative to the node on which the touch listener is registered. Row 4: Return value, double; Method name and argument list, get Y, left parenthesis, right parenthesis, returns the y coordinate of the Touch Point relative to the node on which the touch listener is registered.

Back to Table

A table detailing the handle method of the Event Handler, left angle bracket, T extends Event, right angle bracket interface. void handle, left parenthesis, T, event, right parenthesis. This method is called when an event occurs for which this handler is registered.

Back to Table

A table detailing useful methods of the Node class for mouse and touch events. The table has 4 rows and 2 columns. It has the following column headings. Return value. Method name and argument list. The row details in the table are the following. Row 1: Return value, void; Method name and argument list, set On Mouse Pressed, left parenthesis, Event Handler, left angle bracket, question mark, super, Mouse Event, right angle bracket, handler, right parenthesis, registers handler as the listener to execute when the user presses the mouse. A parameter of null removes any previously registered handler. Row 2: Return value, void; Method name and argument list, set On Mouse Moved, left parenthesis, Event Handler, left angle bracket, question mark, super, Mouse Event, right angle bracket, handler, right parenthesis, registers handler as the listener to execute when the user moves the mouse. A parameter of null removes any previously registered handler. Row 3: Return value, void; Method name and argument list, set On Touch Pressed, left parenthesis, Event Handler, left angle bracket, question mark, super, Touch Event, right angle bracket, handler, right parenthesis, registers handler as the listener to execute when the user touches the screen. A parameter of null removes any previously registered handler. Row 4: Return value, void; Method name and argument list, set On Touch Moved, left parenthesis, Event Handler, left angle bracket, question mark, super, Touch Event, right angle bracket, handler, right parenthesis, registers handler as the listener to execute when the user moves the touch point on the screen. A parameter of null removes any previously registered handler.

Back to Table

Code in the Sub Hunt class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Sub Hunt class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java f x, dot, scene, dot, canvas, dot, Graphics Context, semicolon. Line 6: import, java f x, dot, scene, dot, paint, dot, Color, semicolon. Line 7: import, java, dot, util, dot, Random, semicolon. Line 8: Blank. Line 9: public, class, Sub Hunt. Line 10: Left brace. Line 11, indented once: public, static, i n t, DEFAULT underscore GAME underscore SIZE, equals, 300, semicolon. Line 12, indented once: public, static, i n t, SIDE, equals, 36, semicolon, forward slash, forward slash, size of submarine. Line 13: Blank. Line 14, indented once: private, i n t, game Size, semicolon. Line 15, indented once: private, i n t, x C t r, semicolon, forward slash, forward slash, x coordinate of center of submarine. Line 16, indented once: private, i n t, y C t r, semicolon, forward slash, forward slash, y coordinate of center of submarine. Line 17, indented once: private, String, status, equals, open quotes, close quotes, semicolon. Line 18, indented once: private, boolean, hit, semicolon. Line 19: Blank. Line 20, indented once: Forward slash, asterisk, asterisk, Constructor. Line 21, indented once: Asterisk, at, param, game Size the game size. Line 22, indented once: Asterisk, forward slash. Line 23, indented once: public, Sub Hunt, left parenthesis, i n t, game Size, right parenthesis. Line 24, indented once: Left brace. Line 25, indented twice: if, left parenthesis, game Size, greater than, SIDE, right parenthesis. Line 26, indented 3 times: this, dot, game Size, equals, game Size, semicolon. Line 27, indented twice: else. Line 28, indented 3 times: this, dot, game Size, equals, DEFAULT underscore GAME underscore SIZE, semicolon. Line 29, indented twice: Forward slash, forward slash, generate submarine center. Line 30, indented twice: Random, random, equals, new, Random, left parenthesis, right parenthesis, semicolon. Line 31, indented twice: x C t r, equals, SIDE, forward slash, 2, plus, random, dot, next I n t, left parenthesis, this, dot, game Size, minus, SIDE, right parenthesis, semicolon. Line 32, indented twice: y C t r, equals, SIDE, forward slash, 2, plus, random, dot, next I n t, left parenthesis, this, dot, game Size, minus, SIDE, right parenthesis, semicolon. Line 33, indented twice: hit, equals, false, semicolon. Line 34, indented once: right brace. Line 35: Blank. Line 36, indented once: Forward slash, asterisk, asterisk, get Status Accessor. Line 37, indented once: Asterisk, at, return status. Line 38, indented once: Asterisk, forward slash. Line 39, indented once: public, String, get Status, left parenthesis, right parenthesis. Line 40, indented once: Left brace. Line 41, indented twice: return, status, semicolon. Line 42, indented once: right brace. Line 43: Blank. Line 44, indented once: Forward slash, asterisk, asterisk, get

Game Size Accessor. Line 45, indented once: Asterisk, at, return game Size. Line 46, indented once: Asterisk, forward slash. Line 47, indented once: public, i n t, get Game Size, left parenthesis, right parenthesis. Line 48, indented once: Left brace. Line 49, indented twice: return, game Size, semicolon. Line 50, indented once: Right brace. Line 51: Blank. Line 52, indented once: Forward slash, asterisk, asterisk, is Hit method. Line 53, indented once: Asterisk, at, return hit, which indicates whether the sub has been hit. Line 54, indented once: Asterisk, forward slash. Line 55, indented once: public, boolean, is Hit, left parenthesis, right parenthesis. Line 56, indented once: Left brace. Line 57, indented twice: return, hit, semicolon. Line 58, indented once: right brace. Line 59: Blank. Line 60, indented once: Forward slash, asterisk, asterisk, play method. Line 61, indented once: Asterisk, at, param, x the x coordinate of the play. Line 62, indented once: Asterisk, at, param, y the y coordinate of the play. Line 63, indented once: Asterisk, forward slash. Line 64, indented once: public, void, play, left parenthesis, i n t, x, comma, i n t, y, right parenthesis. Line 65, indented once: Left brace. Line 66, indented twice: Forward slash, forward slash, is click within the submarine? Line 67, indented twice: if, left parenthesis, Math, dot, a b s, left parenthesis, x, minus, x C t r, right parenthesis, less than, SIDE, forward slash, 2. Line 68, indented a few times: ampersand, ampersand, Math, dot, a b s, left parenthesis, y, minus, y C t r, right parenthesis, less than, SIDE, forward slash, 2, right parenthesis. Line

69, indented twice: Left brace. Line 70, indented 3 times: status, equals, open quotes, Sunk!, close quotes, semicolon. Line 71, indented 3 times: hit, equals, true, semicolon. Line 72, indented twice: right brace. Line 73, indented twice: Forward slash, forward slash, is click close? Line 74, indented twice: else, if, left parenthesis, Math, dot, a b s, left parenthesis, x, minus, x C t r, right parenthesis, less than, 2, asterisk, SIDE. Line 75, indented a few times: ampersand, ampersand, Math, dot, a b s, left parenthesis, y, minus, y C t r, right parenthesis, 2, asterisk, SIDE, right parenthesis. Line 76, indented 3 times: status, equals, open quotes, Close, dot, dot, dot, close quotes, semicolon. Line 77, indented twice: Forward slash, forward slash, click is too far from submarine. Line 78, indented twice: else. Line 79, indented 3 times: status, equals, open quotes, In the water, close quotes, semicolon. Line 80, indented once: right brace. Line 81: Blank. Line 82, indented once: Forward slash, asterisk, asterisk, draw method. Line 83, indented once: Asterisk, at, param g c a Graphics Context object. Line 84, indented once: Asterisk, at, param, x the x coordinate of the play. Line 85, indented once: Asterisk, at, param, y the y coordinate of the play. Line 86, indented once: Asterisk, forward slash. Line 87, indented once: public, void, draw, left parenthesis, Graphics Context, g c, comma, i n t, x, comma, i n t, y, right parenthesis. Line 88, indented once: Left brace. Line 89, indented twice: if, left parenthesis, status, dot, equals, left parenthesis, open quotes, Sunk!, close quotes, right parenthesis, right parenthesis. Line 90, indented twice: Left brace. Line 91, indented 3 times: Forward slash, forward slash, draw sunken submarine. Line 92, indented 3 times: g c, dot, set Fill, left parenthesis, Color, dot, BLACK, right parenthesis, semicolon. Line 93, indented 3 times: g c, dot, fill Round Rect, left parenthesis, x C t r, minus, SIDE, forward slash, 2, comma, y C t r, minus, SIDE, forward slash, 2, comma. Line 94, indented a few times: SIDE, forward slash, 2, comma, SIDE, comma, SIDE, forward slash, 2, comma, SIDE, forward slash, 2, right parenthesis, semicolon. Line 95: Blank. Line 96, indented 3 times: g c, dot, fill Round Rect, left parenthesis, x C t r, minus, SIDE, forward slash, 4, comma, y C t r, minus, SIDE, forward slash, 3, comma. Line 97, indented a few times: SIDE, forward slash, 2, comma, SIDE, forward slash, 2, comma, SIDE, forward slash, 4, comma, SIDE, forward slash, 4, right parenthesis, semicolon. Line 98: Blank. Line 99, indented 3 times: g c, dot, stroke Line, left parenthesis, x C t r, plus, SIDE, forward slash, 4, comma, y C t r, minus, SIDE, forward slash, 9, comma. Line 100, indented 3 times: x C t r, plus, SIDE, forward slash, 2, y C t r, minus, SIDE, forward slash, 9, right parenthesis, semicolon. Line 101: Blank. Line 102, indented 3 times: Forward slash, forward slash, draw red depth charge. Line 103, indented 3 times: g c, dot, set Fill, left parenthesis, Color, dot, RED, right parenthesis, semicolon. Line 104, indented 3 times: g c, dot, fill Oval, left parenthesis, x, minus, SIDE, forward slash, 2, comma, y, minus, SIDE, forward slash, 2, comma, SIDE, comma, SIDE, right parenthesis, semicolon. Line 105, indented twice: right brace. Line 106, indented twice: else, if, left parenthesis, status, dot, equals, left parenthesis, open quotes, In the water, close quotes, right parenthesis, right parenthesis, forward slash, forward slash, draw blue circle. Line 107, indented twice: Left brace. Line 108, indented 3 times: g c, dot, set Fill, left parenthesis, Color, dot, BLUE, right parenthesis, semicolon. Line 109, indented 3 times: g c, dot, fill Oval, left parenthesis, x, minus, SIDE, forward slash, 2, comma, y, minus, SIDE, forward slash, 2, comma, SIDE, comma, SIDE, right parenthesis, semicolon. Line 110, indented twice: right brace. Line 111, indented twice: Forward slash, forward slash, else Close, dot, dot, dot, do not draw. Line 112, indented once: right brace. Line 113: right brace.

Back to Example

Code in the Sub Hunt View Controller class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Sub Hunt View Controller class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java f x, dot, scene, dot, layout, dot, asterisk, semicolon. Line 6: import, java f x, dot, scene, dot, input, dot, asterisk, semicolon. Line 7: import, java f x, dot, scene, dot, canvas, dot, Canvas, semicolon. Line 8: import, java f x, dot, scene, dot, canvas, dot, Graphics Context, semicolon. Line 9: import, java f x, dot, stage, dot, Stage, semicolon. Line 10: Blank. Line 11: public, class, Sub Hunt View Controller, extends, V Box. Line 12: Left brace. Line 13, indented once: private, Sub Hunt, sub, semicolon, forward slash, forward slash, submarine. Line 14, indented once: private, Stage, stage, semicolon. Line 15, indented once: private, Graphics Context, g c, semicolon. Line 16: Blank. Line 17, indented once: Forward slash, asterisk, asterisk, Constructor. Line 18, indented once: Asterisk, at, param sub the Model reference for this game. Line 19, indented once: Asterisk, at, param, stage the Stage for this game. Line 20, indented once: Asterisk, at, param, width the width of the Canvas rendering the game. Line 21, indented once: Asterisk, at, param, height the height of the Canvas rendering the game. Line 22, indented once: Asterisk, forward slash. Line 23, indented once: public, Sub Hunt View Controller, left parenthesis, Sub Hunt, sub, comma. Line 24, indented a few times: Stage, stage, comma, i n t, width, comma, i n t, height, right parenthesis. Line 25, indented once: Left brace. Line 26, indented twice: this, dot, sub, equals, sub, semicolon. Line 27, indented once: this, dot, stage, equals, stage, semicolon. Line 28, indented once: Canvas, canvas, equals, new, Canvas, left parenthesis, width, comma, height, right parenthesis, semicolon. Line 29, indented twice: get Children, left parenthesis, right parenthesis, dot, add, left parenthesis, canvas, right parenthesis, semicolon. Line 30, indented once: g c, equals, canvas, dot, get Graphics Context 2 D, left parenthesis, right parenthesis, semicolon. Line 31: Blank. Line 32, indented twice: this, dot, set On Mouse

Clicked, left parenthesis, event, hyphen, right angle bracket, handle Mouse Event, left parenthesis, event, right parenthesis, right parenthesis, semicolon. Line 33, indented twice: this, dot, set On Touch Pressed, left parenthesis, event, hyphen, right angle bracket, handle Touch Event, left parenthesis, event, right parenthesis, right parenthesis, semicolon. Line 34, indented once: right brace. Line 35: Blank. Line 36, indented once: forward slash, asterisk, asterisk, handle Mouse Event method. Line 37, indented once: Asterisk, at, param event the Mouse Event generated by the user. Line 38, indented once: Asterisk, forward slash. Line 39, indented once: public, void, handle Mouse Event, Mouse Event, event, right parenthesis. Line 40, indented once: Left brace. Line 41, indented twice: i n t, x, equals, left parenthesis, i n t, right parenthesis, event, dot, get Scene X, left parenthesis, right parenthesis, semicolon. Line 42, indented twice: i n t, y, equals, left parenthesis, i n t, right parenthesis, event, dot, get Scene Y, left parenthesis, right parenthesis, semicolon. Line 43, indented twice: play, left parenthesis, x, comma, y, right parenthesis, semicolon. Line 44, indented once: right brace. Line 45: Blank. Line 46, indented once: Forward slash, asterisk, asterisk, handle Touch Event method. Line 47, indented once: Asterisk, at, param, event, the Touch Event generated by the user. Line 48, indented once: Asterisk, forward slash. Line 49, indented once: public, void, handle Touch Event, left parenthesis, Touch Event, event, right parenthesis. Line 50, indented once: Left brace. Line 51, indented twice: i n t, x, equals, left parenthesis, i n t, right parenthesis, event, dot, get Touch Point, left parenthesis, right parenthesis, dot, get Scene X, left parenthesis, right parenthesis, semicolon. Line 52, indented twice: i n t, y, equals, left parenthesis, i n t, right parenthesis, event, dot, get Touch Point, left parenthesis, right parenthesis, dot, get Scene Y, left parenthesis, right parenthesis, semicolon. Line 53, indented twice: play, left parenthesis, x, comma, y, right parenthesis, semicolon. Line 54, indented once: right brace. Line 55: Blank. Line 56, indented once: Forward slash, asterisk, asterisk, play method. Line 57, indented once: Asterisk, at, param, x, the x coordinate of the play. Line 58, indented once: Asterisk, at, param, y, the y coordinate of the play. Line 59, indented once: Asterisk, forward slash. Line 60, indented once: public, void, play, left parenthesis, i n t, x, comma, i n t, y, right parenthesis. Line 61, indented once: Left brace. Line 62, indented twice: sub, dot, play, left parenthesis, x, comma, y, right parenthesis, semicolon. Line 63, indented once: sub, dot, draw, left parenthesis, g c, comma, x, comma, y, right parenthesis, semicolon. Line 64, indented once: stage, dot, set Title, left parenthesis, sub, dot, get Status, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 65, indented twice: if, left parenthesis, sub, dot, is Hit, left parenthesis, right parenthesis, right parenthesis. Line 66, indented twice: Left brace. Line 67, indented 3 times: this, dot, set On Mouse Clicked, left parenthesis, null, right parenthesis, semicolon. Line 68, indented 3 times: this, dot, set On

Touch Pressed, left parenthesis, null, right parenthesis, semicolon. Line 69, indented twice: right brace. Line 70, indented once: right brace. Line 71: right brace.

Back to Example

Code in the Play Sub Hunt class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Play Sub Hunt class. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java f x, dot, application, dot, Application, semicolon. Line 6: import, java f x, dot, scene, dot, Scene, semicolon. Line 7: import, java f x, dot, stage, dot, Stage, semicolon. Line 8: Blank. Line 9: public, class, Play Sub Hunt, extends Application. Line 10: Left brace. Line 11, indented once: private, final, i n t, GAME underscore SIZE, equals, 500, semicolon. Line 12: Blank. Line 13, indented once: at, Override. Line 14, indented once: public, void, start, left parenthesis, Stage, stage, right parenthesis. Line 15, indented once: Left brace. Line 16, indented twice: Sub Hunt, sub, equals, new, Sub Hunt, left parenthesis, GAME underscore SIZE, right parenthesis, semicolon. Line 17, indented twice: Sub Hunt View Controller, root, equals. Line 18, indented 3 times: new, Sub Hunt View Controller, left parenthesis, sub, comma, stage, comma, GAME underscore SIZE, comma, GAME underscore SIZE right parenthesis, semicolon. Line 19: Blank. Line 20, indented once: Scene, scene, equals, new, Scene, left parenthesis, root, comma, 500, comma, 500, right parenthesis, semicolon. Line 21, indented twice: stage, dot, set Title, left parenthesis, open quotes, Play, exclamation mark, exclamation mark, close quotes, right parenthesis, semicolon. Line 22, indented once: stage, dot, set Scene, left parenthesis, scene, right parenthesis, semicolon. Line 23, indented twice: stage, dot, show, left parenthesis, right parenthesis, semicolon. Line 24, indented once: right brace. Line 25: Blank. Line 26, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 27, indented once: Left brace. Line 28, indented twice: launch, left parenthesis, a r g s, right parenthesis, semicolon. Line 29, indented once: right brace. Line 30: right brace.

Back to Example

Code in the Treasure Hunt class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Treasure Hunt class. Line 2: Asterisk, Anderson, comma Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java, dot, util, dot, Random, semicolon. Line 6: import, java f x, dot, scene, dot, canvas, dot, Graphics Context, semicolon. Line 7: import, java f x, dot, scene, dot, paint, dot, Color, semicolon. Line 8: import, java f x, dot, scene, dot, text, dot, Font, semicolon. Line 9: Blank. Line 10: public, class, Treasure Hunt. Line 11: Left brace. Line 12, indented once: public, static, i n t, DEFAULT underscore GAME underscore SIZE, equals, 300, semicolon, forward slash, forward slash, side of window. Line 13, indented once:

public, static, i n t, SIDE, equals, 40, semicolon, forward slash, forward slash, side of treasure. Line 14: Blank. Line 15, indented once: private, i n t, game Size, semicolon. Line 16, indented once: private, i n t, x C t r, semicolon, forward slash, forward slash, x coordinate of center of treasure. Line 17, indented once: private, i n t, y C t r, semicolon, forward slash, forward slash, y coordinate of center of treasure. Line 18, indented once: private, String, status, equals, open quotes, close quotes, semicolon, forward slash, forward slash, message. Line 19, indented once: private, boolean, game Over, equals, false, semicolon. Line 20: Blank. Line 21, indented once: forward slash, asterisk, asterisk, Constructor. Line 22, indented once: Asterisk, at, param game Size the size of the game. Line 23: Asterisk, forward slash. Line 24, indented once: public, Treasure Hunt, left parenthesis, i n t, game Size, right parenthesis. Line 25, indented once: Left brace. Line 26, indented twice: if, left parenthesis, game Size, greater than, SIDE, right parenthesis. Line 27, indented 3 times: this, dot, game Size, equals, game Size, semicolon. Line 28, indented twice: else. Line 29, indented 3 times: this, dot, game Size, equals, DEFAULT underscore GAME underscore SIZE, semicolon. Line 30, indented twice: Forward slash, forward slash, generate treasure center. Line 31, indented twice: Random, random, equals, new, Random, left parenthesis, right parenthesis, semicolon. Line 32, indented twice: x C t r, equals, SIDE, forward slash, 2, plus, plus, random, dot, next I n t, left parenthesis, this, dot, game Size, minus, SIDE, right parenthesis, semicolon. Line 33, indented twice: y C t r, equals, SIDE, forward slash, 2, plus, random, dot, next Int, left parenthesis, this, dot, game Size, minus, SIDE, right parenthesis, semicolon. ll. Line 34, indented once: right brace. Line 35: Blank. Line 36, indented once: Forward slash, asterisk, asterisk, get Game Size accessor. Line 37, indented once: Asterisk, at, return game Size. Line 38, indented once: Asterisk, forward slash. Line 39, indented once: public, i n t, get Game Size, left parenthesis, right parenthesis. Line 40, indented once: Left brace. Line 41, indented twice: return, game Size, semicolon. Line 42, indented once: right brace. Line 43: Blank. Line 44, indented once: Forward slash, asterisk, asterisk, is Game Over method. Line 45, indented once: Asterisk, at, return game Over. Line 46, indented once: Asterisk, forward slash. Line 47, indented once: public, boolean, is Game Over, left parenthesis, right parenthesis. Line 48, indented once: Left brace. Line 49, indented twice: return, game Over, semicolon. Line 50, indented once: right brace. Line 51: Blank. Line 52, indented once: Forward slash, asterisk, asterisk, play method. Line 53, indented once: Asterisk, at, param, x, the x coordinate of the play. Line 54, indented once: Asterisk, at, param, y, the y coordinate of the play. Line 55, indented once: Asterisk, forward slash. Line 56, indented once: public, void, play, left parenthesis, i n t, x, comma, i n t, y, right parenthesis. Line 57, indented once: Left brace. Line 58, indented twice: forward slash, forward slash, is mouse within treasure, question mark. Line 59, indented twice: if, left parenthesis, Math, dot, a b s, left parenthesis, x, minus, x C t r, right parenthesis, less than, SIDE, forward slash, 2. Line 60, indented a few times: ampersand, ampersand, Math, dot, a b s, left parenthesis, y, minus, y C t r, right parenthesis, less than, SIDE, forward slash, 2, right parenthesis. Line 61, indented twice: Left brace. Line 62, indented 3 times: game Over, equals, true, semicolon. Line 63, indented 3 times: status, equals, open quotes, Found, close quotes, semicolon. Line 64, indented twice: right brace. Line 65, indented twice: forward slash, forward slash, is mouse within half-length of the treasure, question mark. Line 66, indented twice: else, if, left parenthesis, Math, dot, a b s, left parenthesis, x, minus, x C t r, right parenthesis, less than, left parenthesis, 1.5, asterisk, SIDE, right parenthesis. Line 67, indented a few times: ampersand, ampersand, Math, dot, a b s, left parenthesis, y, minus, y C t r, right parenthesis, less than, left parenthesis, 1.5, asterisk, SIDE, right parenthesis, right parenthesis. Line 68, indented 3 times: status, equals, open quotes, Hot, close quotes, semicolon. Line 69, indented twice: forward slash, forward slash, is mouse within 1 length of the treasure, question mark. Line 70, indented twice: else, if, left parenthesis, Math, dot, a b s, left parenthesis, x, minus, x C t r, right parenthesis, less than, left parenthesis, 2, asterisk, SIDE, right parenthesis. Line 71, indented a few times: ampersand, ampersand, Math, dot, a b s, left parenthesis, y, minus, y C t r, right parenthesis, less than, left parenthesis, 2, asterisk, SIDE, right parenthesis, right parenthesis. Line 72, indented 3 times: status, equals, open quotes, Warm, close quotes, semicolon. Line 73, indented twice: forward slash, forward slash, is mouse within 2 lengths of the treasure, question mark. Line 74, indented twice: else, if, left parenthesis, Math, dot, a b s, left parenthesis, x, minus, x C t r, right parenthesis, less than, left parenthesis, 3, asterisk, SIDE, right parenthesis. Line 75, indented a few times: ampersand, ampersand, Math, dot, a b s, left parenthesis, y, minus, y C t r, right parenthesis, less than, left parenthesis, 3, asterisk, SIDE, right parenthesis, right parenthesis. Line 76, indented 3 times: status, equals, open quotes, Lukewarm, close quotes, semicolon. Line 77, indented twice: else, forward slash, forward slash, mouse is not near treasure. Line 78, indented 3 times: status, equals, open quotes, Cold, close quotes, semicolon. Line 79, indented once: right brace. Line 80: Blank. Line 81, indented once: forward slash, asterisk, asterisk, draw method. Line 82, indented once: Asterisk, at, param, g c a Graphics Context reference. Line 83, indented once: Asterisk, at, param, x, the x coordinate of the play. Line 84, indented once: Asterisk, at, param, y, the y coordinate of the play. Line 85, indented once: Asterisk, forward slash. Line 86, indented once: public, void, draw, left parenthesis, Graphics Context, g c, comma, i n t, x, comma, i n t, y, right parenthesis. Line 87, indented once: Left brace. Line 88, indented twice: g c, dot, set Fill, left parenthesis, Color, dot, BLUE, right parenthesis, semicolon. Line 89, indented twice: g c, dot, set Font, left parenthesis, new, Font, left parenthesis, 24, right parenthesis, right parenthesis, semicolon. Line 90, indented twice: g c, dot, clear Rect, left parenthesis, 0, comma, 0, comma, game Size, comma, game Size, right parenthesis, semicolon. Line 91, indented twice: if, left parenthesis, status, dot, equals, left parenthesis, open quotes, Found, close quotes, right parenthesis, right parenthesis, forward slash, forward slash, if found, draw treasure. Line 92, indented twice: Left brace. Line 93, indented 3 times: g c, dot, set Font, left parenthesis, new, Font, left parenthesis, 16, right parenthesis, right parenthesis, semicolon. Line 94, indented 3 times: g c, dot, set Fill, left parenthesis, Color, dot, RED, right parenthesis, semicolon. Line 95, indented 3 times: g c, dot, fill Rect, left parenthesis, x C t r, minus, SIDE, forward slash, 2, comma, y C t r, minus, SIDE, forward slash, 2, comma, SIDE, comma, SIDE, right parenthesis, semicolon. Line 96, indented 3 times: g c, dot, set Fill, left parenthesis, Color, dot, GREEN, right parenthesis, semicolon. Line 97, indented 3 times: g c, dot, fill Text, left parenthesis, open quotes, dollar, dollar, dollar, close quotes, comma, x C t r, minus, SIDE, forward slash, 3, comma, y C t r, plus, SIDE, forward slash, 6, right parenthesis, semicolon. Line 98, indented twice: right brace. Line 99, indented twice: else. Line 100, indented 3 times: g c, dot, fill Text, left parenthesis, status, comma, x, comma, y, right parenthesis, semicolon, forward slash, forward slash, display current status. Line 101, indented once: right brace. Line 102: right brace.

Back to Example

Code in the Treasure Hunt View Controller class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Treasure Hunt View Controller class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java f x, dot, event, dot, asterisk, semicolon. Line 6: import, java f x, dot, scene, dot, layout, dot, asterisk, semicolon. Line 7: import, java f x, dot, scene, dot, input, dot, asterisk, semicolon. Line 8: import, java f x, dot, scene, dot, canvas, dot, Canvas, semicolon. Line 9: import, java f x, dot, scene, dot, canvas, dot, Graphics Context, semicolon. Line 10: import, java f x, dot, stage, dot, Stage, semicolon. Line 11: Blank. Line 12: public, class, Treasure Hunt View Controller, extends, V Box. Line 13, indented a few times: implements, Event Handler, left angle bracket, Input Event, right angle bracket. Line 14: Left brace. Line 15, indented once: private, int width, height, semicolon. Line 16, indented once: private, Treasure Hunt, hunt, semicolon. Line 17, indented once: private, Graphics Context, g c, semicolon. Line 18: Blank. Line 19, indented once: Forward slash, asterisk, asterisk, Constructor. Line 20, indented once: Asterisk, at, param, hunt the model reference. Line 21, indented once: Asterisk, at, param, width the width of the Canvas rendering the game. Line 22, indented once: Asterisk, at, param, height the height of the Canvas rendering the game. Line 23, indented once: Asterisk, forward slash. Line 24, indented once: public, Treasure Hunt View Controller, left parenthesis, Treasure Hunt, hunt, comma, i n t, width, comma, i n t, height, right parenthesis. Line 25, indented once: Left brace. Line 26, indented twice: this, dot, hunt, equals, hunt, semicolon. Line 27, indented twice: this, dot, width, equals, width, semicolon. Line 28, indented twice: this, dot, height, equals, height, semicolon. Line 29, indented twice: Canvas, canvas, equals, new, Canvas, left parenthesis, width, comma, height, right parenthesis, semicolon. Line 30, indented twice: get Children, left parenthesis, right parenthesis, dot, add, left parenthesis, canvas, right parenthesis, semicolon. Line 31, indented twice: g c, equals, canvas, dot, get Graphics Context 2 D, left parenthesis, right parenthesis, semicolon. Line 32: Blank. Line 33, indented twice: this, dot, set On Mouse Moved, left parenthesis, this, right parenthesis, semicolon. Line 34, indented twice: this, dot, set On Touch Moved, left parenthesis, this, right parenthesis, semicolon. Line 35, indented once: right brace. Line 36: Blank. Line 37, indented once: forward slash, asterisk, asterisk, handle method. Line 38, indented once: Asterisk, at, param, event, the Input Event generated by the user. Line 39, indented once: Asterisk, forward slash. Line 40, indented once: public, void, handle, left parenthesis, Input Event, event, right parenthesis. Line 41, indented once: left brace. Line 42, indented twice: i n t, x, equals, minus, 100, comma, y, equals, minus, 100, semicolon. Line 43, indented twice: if, left parenthesis, event, dot, get Event

Type, left parenthesis, right parenthesis, equals, equals, Mouse Event, dot, MOUSE underscore MOVED, right parenthesis. Line 44, indented twice: Left brace. Line 45, indented 3 times: Mouse Event, m Event, equals, left parenthesis, Mouse Event, right parenthesis, event, semicolon. Line 46, indented 3 times: x, equals, left parenthesis, i n t, right parenthesis, m Event, dot, get Scene X, left parenthesis, right parenthesis, semicolon. Line 47, indented a 3 times: y, equals, left parenthesis, i n t, right parenthesis, m Event, dot, get Scene Y, left parenthesis, right parenthesis, semicolon. Line 48, indented twice: right brace. Line 49, indented twice: else, if, left parenthesis, event, dot, get Event Type, left parenthesis, right parenthesis, equals, equals, Touch Event, dot, TOUCH underscore MOVED, right parenthesis. Line 50, indented twice: Left brace. Line 51, indented 3 times: Touch Event, t Event, equals, left parenthesis, Touch Event, right parenthesis, event, semicolon. Line 52, indented 3 times: x, equals, left parenthesis, i n t, right parenthesis, t Event, dot, get Touch Point, left parenthesis, right parenthesis, dot, get Scene X, left parenthesis, right parenthesis, semicolon. Line 53, indented 3 times: y, equals, left parenthesis, i n t, right parenthesis, t Event, dot, get Touch Point, left parenthesis, right parenthesis, dot, get Scene Y, left parenthesis, right parenthesis, semicolon. Line 54, indented twice: right brace. Line 55: Blank. Line 56, indented twice: hunt, dot, play, left parenthesis, x, comma, y, right parenthesis, semicolon. Line 57: Blank. Line 58, indented twice: hunt, dot, draw, left parenthesis, g c, comma, x, comma, y, right parenthesis, semicolon. Line 59, indented twice: if, left parenthesis, hunt, dot, is Game Over, left parenthesis, right parenthesis, right parenthesis. Line 60, indented twice: Left brace. Line 61, indented 3 times: this, dot, set On Mouse Moved, left parenthesis, null, right parenthesis, semicolon. Line 62, indented 3 times: this, dot, set On Touch Moved, left parenthesis, null, right parenthesis, semicolon. Line 63, indented 3 times: Stage, stage, equals, left parenthesis, Stage, right parenthesis, get Scene, left parenthesis, right parenthesis, dot, get Window, left parenthesis, right parenthesis, semicolon. Line 64, indented 3 times: stage, dot, set Title, left parenthesis, open quotes, Thank you for playing, close quotes, right parenthesis, semicolon. Line 65, indented twice: right brace. Line 66, indented once: right brace. Line 67: right brace.

Back to Example

A table detailing useful Event Type constants of the Mouse Event and Touch Event classes. The table has 2 rows and 2 columns. It has the following column headings. Class. Constant. The row details in the table are the following. Row 1: Class, Mouse Event; Constant, MOUSE Underscore MOVED constant identifying a mouse move event. Row 2: Class, Touch Event; Constant, TOUCH underscore MOVED constant identifying a touch move event.

Back to Table

A table detailing useful classes and methods to retrieve the stage. The table has 2 rows and 2 columns. It has the following column headings. Class. Method. Row 1: Class, Node; Method, Scene, get Scene, left parenthesis, right parenthesis, returns the Scene containing this Node. Row 2: Class, Scene; Method, Window, get Window, left parenthesis, right parenthesis, returns the window for this Scene.

Back to Table

Code in the Play Treasure Hunt class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Play Treasure Hunt class. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java f x, dot, application, dot, Application, semicolon. Line 6: import, java f x, dot, scene, dot, Scene, semicolon. Line 7: import, java f x, dot, stage, dot, Stage, semicolon. Line 8: Blank. Line 9: public, class, Play Treasure Hunt, extends, Application. Line 10: Left brace. Line 11, indented once: private, final, i n t, GAME underscore SIZE, equals, 500, semicolon. Line 12: Blank. Line 13, indented once: at, Override. Line 14, indented once: public, void, start, left parenthesis, Stage, stage, right parenthesis. Line 15, indented once: Left brace. Line 16, indented twice: Treasure Hunt, t h, equals, new, Treasure Hunt, left parenthesis, GAME underscore SIZE, right parenthesis, semicolon. Line 17, indented twice: Treasure Hunt View Controller, root, equals. Line 18, indented 3 times: new, Treasure Hunt View Controller, left parenthesis, t h, comma, GAME underscore SIZE, comma, GAME underscore SIZE, right parenthesis, semicolon. Line 19: Blank. Line 20, indented twice: Scene, scene, equals, new, Scene, left parenthesis, root, comma, GAME underscore SIZE, comma, GAME underscore SIZE, right parenthesis, semicolon. Line 21, indented twice: stage, dot, set Title, left parenthesis, open quotes, Play, exclamation mark, exclamation mark, close quotes, right parenthesis, semicolon. Line 22, indented twice: stage, dot, set Scene, left parenthesis, scene, right parenthesis, semicolon. Line 23, indented twice: stage, dot, show, left parenthesis, right parenthesis, semicolon. Line 24, indented once: right brace. Line 25: Blank. Line 26, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 27, indented once: left brace. Line 28, indented twice: launch, left parenthesis, a r g s, right parenthesis, semicolon. Line 29, indented once: right brace. Line 30: right brace.

Back to Example

A table detailing Pie Chart dot Data and Pie Chart constructors. The package is java f x dot scene dot chart. The constructors the following. Pie Chart dot Data, left parenthesis, String, name, comma, double, value, right parenthesis, creates a Pie Chart dot Data, a slice of a Pie

Chart. The slice value is value and its label is name. Pie Chart, left parenthesis, Observable List, left angle bracket, Pie Chart dot Data, right angle bracket, data, right parenthesis, creates a Pie Chart using data as its data for all its slices.

Back to Table

The program code of an application to display a pie chart. Program code. In the code, the words in the variable names are merged. Line 1: import, java, dot, util, dot, Array List, semicolon. Line 2: import, java f x, dot, application, dot, Application, semicolon. Line 3: import, java f x, dot, collections, dot, F X Collections, semicolon. Line 4: import, java f x, dot, collections, dot, Observable List, semicolon. Line 5: import, java f x, dot, scene, dot, chart, dot, asterisk, semicolon. Line 6: import, java f x, dot, scene, dot, Scene, semicolon. Line 7: import, java f x, dot, stage, dot, Stage, semicolon. Line 8: Blank. Line 9: public, class, Monthly Budget Pie Chart, extends, Application. Line 10: Left brace. Line 11, indented once: private, String, left parenthesis, right parenthesis, categories. Line 12, indented twice: equals, left brace, open quotes, Rent, close quotes, comma, open quotes, Food, close quotes, comma, open quotes, Phone, close quotes, comma, open quotes, Entertainment, close quotes, comma, open quotes, Utilities, close quotes, right brace, semicolon. Line 13, indented once: private, double, left square brace, right square brace, expenses. Line 14, indented twice: equals, left brace, 800, comma, 450, comma, 72.50, comma, 200, comma, 90, right brace, semicolon. Line 15: Blank. Line 16, indented once: at, Override. Line 17, indented once: public, void, start, left parenthesis, Stage, stage, right parenthesis. Line 18, indented once: Left brace. Line 19, indented twice: Forward slash, forward slash, Create a Pie Chart, dot, Data of monthly expenses. Line 20, indented twice: Array List, left angle bracket, Pie Chart, dot, Data, right angle bracket, list, equals, new, Array List, left angle bracket, Pie Chart, dot, Data, right angle bracket, left parenthesis, right parenthesis, semicolon. Line 21, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, expenses, dot, length, semicolon, i, plus, plus, right parenthesis. Line 22, indented 3 times: list, dot, add, left parenthesis, new, Pie Chart, dot, Data, left parenthesis, categories, left square brace, i, right square brace, comma, expenses, left square brace, i, right square brace, right parenthesis, right parenthesis, semicolon. Line 23, indented twice: Observable List, less than, Pie Chart, dot, Data, right angle bracket, pie Chart Data. Line 24, indented 3 times: equals, F X Collections, dot, observable List, left parenthesis, list, right parenthesis, semicolon. Line 25: Blank. Line 26, indented twice: Forward slash, forward slash, Create a Pie Chart with pie Chart Data. Line 27, indented twice: Pie Chart, chart, equals, new, Pie Chart, left parenthesis, pie Chart Data, right parenthesis, semicolon. Line 28, indented twice: chart, dot, set Title, left parenthesis, open quotes, Monthly Budget, close quotes, right parenthesis, semicolon. Line

29: Blank. Line 30, indented twice: Forward slash, forward slash, Define the scene and stage. Line 31, indented twice: Scene, scene, equals, new, Scene, left parenthesis, chart, comma, 500, comma, 400, right parenthesis, semicolon. Line 32, indented twice: stage, dot, set Scene, left parenthesis, scene, right parenthesis, semicolon. Line 33, indented twice: stage, dot, set Title, left parenthesis, open quotes, Pie Chart for Monthly Expenses, close quotes, right parenthesis, semicolon. Line 34, indented twice: stage, dot, show, left parenthesis, right parenthesis, semicolon. Line 35, indented once: right brace. Line 36: Blank. Line 37, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 38, indented once: Left brace. Line 39, indented twice: launch, left parenthesis, a r g s, right parenthesis, semicolon. Line 40, indented once: right brace. Line 41: right brace.

Back to Example

A table detailing the observable List method of the F X collections class. Return value: Observable, List, left angle bracket, E, right angle bracket. Method name and argument list: observable, left parenthesis, List, left angle bracket, E, right angle bracket, list, right parenthesis, constructs and returns an Observable List composed of elements of type E from list. Changes to that Observable List will be reflected in components that are registered on that Observable List.

Back to Table

Code for the Investment class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, abstract Investment class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: public, abstract, class, Investment. Line 6: Left brace. Line 7, indented once: public, static, final, i n t, DEFAULT underscore YEARS, equals, 4, semicolon. Line 8, indented once: public, static, final, double, BEGIN underscore INVEST underscore AMOUNT, equals, 100, semicolon. Line 9, indented once: private, double, left square brace, right square brace, amounts, semicolon. Line 10, indented once: private, i n t, years, semicolon. Line 11, indented once: private, double, growth Rate, semicolon. Line 12, indented once: private, String, type, semicolon. Line 13: Blank. Line 14, indented once: Forward slash, asterisk, Constructor. Line 15, indented once: Asterisk, at, param, years, an i n t, the new value of years. Line 16, indented once: Asterisk, at, param type, a String, the new value for the investment type. Line 17, indented once: Asterisk, forward slash. Line 18, indented once: public, Investment, left parenthesis, i n t, years, comma, String, type, right parenthesis. Line 19, indented once: Left brace. Line 20, indented twice: if, left parenthesis, years, less than, 1, right parenthesis. Line 21, indented 3 times: this, dot, years, equals, DEFAULT underscore YEARS, semicolon. Line 22, indented twice:

else. Line 23, indented 3 times: this, dot, years, equals, years, semicolon. Line 24: Blank. Line 25, indented twice: amounts, equals, new, double, left square brace, this, dot, years, right square brace, semicolon. Line 26, indented twice: amounts, left square brace, 0, right square brace, equals, BEGIN underscore INVEST underscore AMOUNT, semicolon. Line 27: Blank. Line 28, indented twice: this, dot, type, equals, type, semicolon. Line 29, indented once: right brace. Line 30: Blank. Line 31, indented once: Forward slash, asterisk, get Amounts method. Line 32, indented once: Asterisk, at, return a copy of amounts, a double, left square brace, right square brace. Line 33, indented once: Asterisk, forward slash. Line 34, indented once: public, double, left square brace, right square brace, get Amounts, left parenthesis, right parenthesis. Line 35, indented once: Left brace. Line 36, indented twice: double, left square brace, right square brace, temp, equals, new, double, left square brace, amounts, dot, length, right square brace, semicolon. Line 37, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, amounts, dot, length, semicolon, i, plus, plus, right parenthesis. Line 38, indented a few times: temp, left square brace, i, right square brace, equals, amounts, left square brace, i, right square brace, semicolon. Line 39, indented twice: return, temp, semicolon. Line 40, indented once: right brace. Line 41: blank. Line 42, indented once: Forward slash, asterisk, Accessor method for years. Line 43, indented once: Asterisk, at, return number of years. Line 44, indented once: Asterisk, forward slash. Line 45, indented once: public, i n t, get Years, left parenthesis, right parenthesis. Line 46, indented once: Left brace. Line 47, indented once: return, years, semicolon. Line 48, indented once: right brace. Line 49: Blank. Line 50, indented once: Forward slash, asterisk, Accessor method for type. Line 51, indented once: Asterisk, at, return investment type. Line 52, indented once: Asterisk, forward slash. Line 53, indented once: public, String, get Type, left parenthesis, right parenthesis. Line 54, indented once: Left brace. Line 55, indented twice: return, type, semicolon. Line 56, indented once: right brace. Line 57: Blank. Line 58, indented once: Forward slash, asterisk, Accessor method for growth Rate. Line 59, indented once: Asterisk, at, return growth Rate. Line 60, indented once: Asterisk, forward slash. Line 61, indented once: public, double, get Growth Rate, left parenthesis, right parenthesis. Line 62, indented once: Left brace. Line 63, indented twice: return, growth Rate, semicolon. Line 64, indented once: right brace. Line 65: Blank. Line 66, indented once: Forward slash, asterisk, Mutator method for growth Rate. Line 67, indented once: Asterisk, at, param growth Rate, the new value for growth Rate. Line 68, indented once: Asterisk, at, return a reference to this object. Line 69, indented once: Asterisk, forward slash. Line 70, indented once: public, Investment, set Growth Rate, left parenthesis, double, growth Rate, right parenthesis. Line 71, indented once: Left brace. Line 72, indented twice: this, dot, growth Rate, equals, growth Rate, semicolon. Line 73, indented twice: return, this, semicolon. Line 74, indented once: right brace. Line 75: Blank. Line 76, indented once: forward slash, asterisk, update Amounts method. Line 77, indented once: Asterisk, updates amounts based on growth Rate. Line 78, indented once: Asterisk, forward slash. Line 79, indented once: public, void, update Amounts, left parenthesis, right parenthesis. Line 80, indented once: Left brace. Line 81, indented twice: for, left parenthesis, i n t, i, equals, 1, semicolon, i, less than, amounts, dot, length, semicolon, i, plus, plus, right parenthesis. Line 82, indented 3 times: amounts, left parenthesis, i, right parenthesis, equals, amounts, left square brace, i, minus, 1, right square brace, asterisk, left parenthesis, 1, plus, growth Rate, right parenthesis, semicolon. Line 83, indented once: right brace. Line 84: Blank. Line 85, indented once: forward slash, asterisk, update abstract method. Line 86, indented once: Asterisk, forward slash. Line 87, indented once: public, abstract, void, update, left parenthesis, right parenthesis, semicolon. Line 88: right brace.

Back to Example

Code for the Bonds class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Bonds class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: public, class, Bonds, extends, Investment. Line 6: Left brace. Line 7, indented once: private, final, double, BOND underscore GROWTH underscore RATE, equals, .03, semicolon. Line 8: Blank. Line 9, indented once: Forward slash, asterisk, Constructor. Line 10, indented once: Asterisk, at, param years the new number of years. Line 11, indented once: asterisk, forward slash. Line 12, indented once: public, Bonds, left parenthesis, i n t, years, right parenthesis. Line 13, indented once: Left brace. Line 14, indented twice: super, left parenthesis, years, comma, open quotes, Bonds, close quotes, right parenthesis, semicolon. Line 15, indented twice: set Growth Rate, left parenthesis, BOND underscore GROWTH underscore RATE, right parenthesis, semicolon. Line 16, indented twice: update Amounts, left parenthesis, right parenthesis, semicolon. Line 17, indented once: right brace. Line 18: Blank. Line 19, indented once: Forward slash, asterisk, update method. Line 20, indented once: Asterisk, do nothing method: Leaves the growth rate unchanged. Line 21, indented once: Asterisk, forward slash: Line 22, indented once: public, void, update, left parenthesis, right parenthesis. Line 23, indented once: Left brace. Line 24, indented twice: forward slash, forward slash, growth rate does not change. Line 25, indented once: right brace. Line 26: right brace.

Back to Example

Code for the Stocks class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Stocks class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: public, class, Stocks, extends, Investment. Line 6: Left brace. Line 7, indented once: private, static, final, double, SLOW underscore GROWTH underscore RATE, equals, .02, semicolon. Line 8, indented once: private, static, final, double, FAST underscore GROWTH underscore RATE, equals, .07, semicolon. Line 9: Blank. Line 10, indented once: forward slash, asterisk, Constructor. Line 11, indented once: asterisk, at, param years the new number of years. Line 12, indented once: Asterisk, forward slash. Line 13, indented once: public, Stocks, left parenthesis, i n t, years, right parenthesis. Line 14, indented once: Left brace. Line 15, indented twice: super, left parenthesis, years, comma, open quotes, Stocks, close quotes, right parenthesis, semicolon. Line 16, indented once: set Growth Rate, left parenthesis, SLOW underscore GROWTH underscore RATE, right parenthesis, semicolon. Line 17, indented once: update Amounts, left parenthesis, right parenthesis, semicolon. Line 18, indented once: right brace. Line 19: Blank. Line 20, indented once: forward slash, asterisk, update method. Line 21, indented once: Asterisk, toggles the growth rate and updates amounts accordingly. Line 22, indented once: Asterisk, forward slash. Line 23, indented once: public, void, update, left parenthesis, right parenthesis. Line 24, indented once: Left brace. Line 25, indented twice: if, left parenthesis, Math, dot, a b s, left parenthesis, get Growth Rate, left parenthesis, right parenthesis, minus, SLOW underscore GROWTH underscore RATE, right parenthesis, less than, 0.0001, right parenthesis. Line 26, indented 3 times: set Growth Rate, left parenthesis, FAST underscore GROWTH underscore RATE, right parenthesis, semicolon. Line 27, indented twice: else. Line 28, indented 3 times: set Growth Rate, left parenthesis, SLOW underscore GROWTH underscore RATE, right parenthesis, semicolon. Line 29: Blank. Line 30, indented twice: update Amounts, left parenthesis, right parenthesis, semicolon. Line 31, indented once: right brace. Line 32: right brace.

Back to Example

Code for the View class that displays a bar chart. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Bar Chart View class. Line 2: Asterisk, Layout using a Border Pane's. Line 3: Asterisk, bottom and center positions. Line 4: Asterisk, The bottom H Box is made up of one button. Line 5: Asterisk, The center position is made up of a Bar Chart. Line 6: Asterisk, Anderson, Franceschi. Line 7: Asterisk, forward slash. Line 8: Blank. Line 9: import, java, dot, util, dot, Array List, semicolon. Line 10: import, java f x, dot, event, dot, asterisk, semicolon. Line 11: import, java f x, dot, geometry, dot, P o s, semicolon. Line 12: import, java f x, dot, scene, dot, control, dot, asterisk, semicolon. Line 13: import, java f x, dot, scene, dot, layout, dot, asterisk, semicolon. Line 14: import, java f x, dot, scene, dot, chart, dot, asterisk, semicolon. Line 15: Blank. Line 16: public, class, Bar Chart View, extends, Border Pane. Line 17: Left brace. Line 18, indented once: private, H Box, button Panel, semicolon, forward slash, forward slash, bottom, holds button. Line 19, indented once: private, Button, button, semicolon. Line 20, indented once: private, Bar Chart, left angle bracket, String, comma, Number, right angle bracket, b c, semicolon, forward slash, forward slash, center, holds chart. Line 21: Blank. Line 22, indented once: Array List, left angle bracket, X Y Chart, dot, Series, left angle bracket, String, comma, Number, right angle bracket, right angle bracket, series, semicolon. Line 23, indented once: private, i n t, starting Year, semicolon. Line 24: Blank. Line 25, indented once: public, Bar Chart View, left parenthesis, i n t, starting Year, comma, Investment, left square brace, right square brace, investments, right parenthesis. Line 26, indented once: Left brace. Line 27, indented twice: super, left parenthesis, right parenthesis, semicolon. Line 28, indented twice: this, dot, starting Year, equals, starting Year, semicolon. Line 29: Blank. Line 30, indented twice: button Panel, equals, new, H Box, left parenthesis, right parenthesis, semicolon. Line 31, indented twice: button, equals, new, Button, left parenthesis, right parenthesis, semicolon. Line 32, indented twice: button, dot, set Text, left parenthesis, open quotes, NEXT SCENARIO, close quotes, right parenthesis, semicolon. Line 33, indented twice: button Panel, dot, set Alignment, left parenthesis, P o s, dot, CENTER, right parenthesis, semicolon. Line 34, indented twice: button Panel, dot, get Children, left parenthesis, right parenthesis, dot, add, left parenthesis, button, right parenthesis, semicolon. Line 35, indented twice: set Bottom, left parenthesis, button Panel, right parenthesis, semicolon. Line 36: Blank. Line 37, indented twice: add Bar Chart, left parenthesis, investments, right parenthesis, semicolon. Line 38, indented once: right brace. Line 39: Blank. Line 40, indented once: public, void, add Bar Chart, left parenthesis, Investment, left square brace, right square brace, investments, right parenthesis. Line 41, indented once: Left brace. Line 42, indented twice: forward slash, forward slash, Create a Bar Chart for investment. Line 43, indented twice: Category Axis, x Axis, equals, new, Category Axis, left parenthesis, right parenthesis, semicolon. Line 44, indented twice: Number Axis, y Axis, equals, new, Number Axis, left parenthesis, right parenthesis, semicolon. Line 45, indented twice: b c, equals, new, Bar Chart, left angle bracket, String, comma, Number, right angle bracket, left parenthesis, x Axis, comma, y Axis, right parenthesis, semicolon. Line 46, indented twice: b c, dot, set Title, left parenthesis, open quotes, Investment Scenarios, close quotes, right parenthesis, semicolon. Line 47, indented twice: x Axis, dot, set Label, left parenthesis, open quotes, Year, close quotes, right parenthesis, semicolon. Line 48, indented twice: y Axis, dot, set Label, left parenthesis, open quotes, Value, close quotes, right parenthesis, semicolon. Line 49: Blank. Line 50, indented twice: Forward slash, forward slash, Create an Array List of X Y Chart, dot, Series using investments data. Line 51, indented twice: series, equals, new, Array List, left angle bracket, X Y Chart, dot, Series, left angle bracket, String, comma, Number, right angle bracket, right angle bracket, left parenthesis, right parenthesis, semicolon. Line 52: Blank. Line 53, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, investments, dot, length, semicolon, i, plus, plus, right parenthesis. Line 54, indented twice: left brace. Line 55, indented 3 times: X Y Chart, dot, Series, left angle bracket, String, comma, Number, right angle bracket, current Investment Series. Line 56, indented 4 times: equals, new, X Y Chart, dot, Series, left angle bracket, String, comma, Number, right angle bracket, left parenthesis, right parenthesis, semicolon. Line 57, indented 3 times: current Investment Series, dot, set Name, left parenthesis, investments, left square brace, i, right square brace, dot, get Type, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 58: Blank. Line 59, indented 3 times: double, left square brace, right square brace, current Amounts, equals, investments, left square brace, i, right square brace, dot, get Amounts, left parenthesis, right parenthesis, semicolon. Line 60, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, current Amounts, dot, length, semicolon, j, plus, plus, right parenthesis. Line 61, indented 3 times: Left brace. Line 62, indented 4 times: current Investment Series, dot, get Data, left parenthesis, right parenthesis, dot, left parenthesis. Line 63, indented 5 times: new, X Y Chart, dot, Data, left angle bracket, String, comma, Number, right angle bracket, left parenthesis. Line 64, indented 6 times: String, dot, value Of, left parenthesis, starting Year, plus, j, right parenthesis, comma, current Amounts, left square brace, j, right square brace, right parenthesis, right parenthesis, semicolon. Line 65, indented 3 times: right brace. Line 66, indented 3 times: series, dot, add, left parenthesis, current Investment Series, right parenthesis, semicolon. Line 67, indented twice: right brace. Line 68, indented twice: forward slash, forward slash, Fill Bar Chart with X Y Chart, dot, Series array. Line 69, indented twice: b c, dot, get Data, left parenthesis, right parenthesis, dot, add All, left parenthesis, series, right parenthesis, semicolon. Line 70, indented twice: set Center, left parenthesis, b c, right parenthesis, semicolon. Line 71, indented once: right brace. Line 72: Blank. Line 73, indented once: public, void, update, left parenthesis, Investment, left square brace, right square brace, investments, right parenthesis. Line 74, indented once: Left brace. Line 75, indented twice: forward slash, forward slash, update investments. Line 76, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, investments, dot, length, semicolon, i, plus, plus, right parenthesis. Line 77, indented twice: Left brace. Line 78, indented 3 times: forward slash, forward slash, do not update bonds series. Line 79, indented 3 times: if, left parenthesis, exclamation mark, investments, left square brace, i, right square brace, dot, get Type, left parenthesis, right parenthesis, equals, left parenthesis, open quotes, Bonds, close quotes, right parenthesis, right parenthesis. Line 80, indented 3 times: Left brace. Line 81, indented 4 times: forward slash, forward slash, retrieve series. Line 82, indented 4 times: X Y Chart, dot, Series, left angle bracket, String, comma, Number, right angle bracket, current Investment Series> line 83, indented 5 times: equals, series, dot, get, left parenthesis, i, right parenthesis, semicolon. Line 84: Blank. Line 85, indented 4 times: forward slash, forward slash, change data of series. Line 86, indented 4 times: current Investment Series, dot, get Data, left parenthesis, right parenthesis, dot, clear, left parenthesis, right parenthesis, semicolon. Line 87, indented 4 times: double, left square brace, right square brace, current Amounts, equals, investments, left square brace, i, right square brace, dot, get Amounts, left parenthesis, right parenthesis, semicolon. Line 88, indented 4 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, current Amounts, dot, length, semicolon, j, plus, plus, right parenthesis. Line 89, indented 4 times: Left brace. Line 90, indented 5 times: current Investment Series, dot, get Data, left parenthesis, right parenthesis, dot, add, left parenthesis. Line 91, indented 6 times: new, X Y Chart, dot, Data, left angle bracket, String, comma, Number, right angle bracket, left parenthesis, open quotes, close quotes. Line 92, indented 7 times: plus, left parenthesis, starting Year, plus, j, right parenthesis, comma, current Amounts, left square bracket, j, right square bracket, right parenthesis, right parenthesis, semicolon. Line 93, indented 4 times: right brace. Line 94: Blank. Line 95, indented 4 times:

forward slash, forward slash, update series. Line 96, indented 4 times: series, dot, set, left parenthesis, i, comma, current Investment Series, right parenthesis, semicolon. Line 97, indented 3 times: right brace. Line 98, indented twice: right brace. Line 99, indented once: right brace. Line 100: Blank. Line 101, indented once: public, void, set Button Listener, left parenthesis, Event Handler, left angle bracket, Action Event, right angle bracket, e h, right parenthesis. Line 102, indented once: Left brace. Line 103, indented twice: button, dot, set On Action, left parenthesis, e h, right parenthesis, semicolon. Line 104, indented once: right brace. Line 105: right brace.

Back to Example

A table detailing a constructor of the Bar Chart class. The Bar Chart class belongs to the package java f x dot scene dot chart. Bar Chart, left parenthesis, Axis, left angle bracket, X, right angle bracket, x Axis, comma, Axis, left angle bracket, Y, right angle bracket, y Axis, right parenthesis, creates a Bar Chart with the two axes. One of the axes should be a Category Axis, the other one a Value Axis. Their order determines if the bar chart is horizontal or vertical.

Back to Table

A table detailing X Y Chart dot Series and X Y Chart dot Data constructors. The package is java f x, dot, scene, dot, chart. The constructors are the following. X Y Chart, dot, Series, left parenthesis, right parenthesis, creates an X Y Chart dot Series that will contain X Y Chart dot Data items. X Y Chart, dot, Data, left parenthesis, X, x Value, comma, Y, y Value, right parenthesis, creates data for the two axes of the chart; the X and Y types should match the X and Y types for the x and y axes.

Back to Table

A table detailing the get Data and set Name methods of the X Y Chart dot Series class. The table has 2 rows and 2 columns. It has the following column headings. Return value. Method name and argument list. The row details are the following. Row 1: Return value, Observable List, left angle bracket, X Y Chart, dot, Data, left angle bracket, X, comma, Y, right angle bracket, right angle bracket; Method name and argument list, get Data, left parenthesis, right parenthesis, returns the Observable List of data items that make up this series. Row 2: Return value, void; Method name and argument list, set Name, left parenthesis, String, value, right parenthesis, sets the name to be displayed for this series.

Back to Table

Code for the controller in the Bar Chart application. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Bar Chart View Controller class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java f x, dot, event, dot, asterisk, semicolon. Line 6: import, java f x, dot, scene, dot, control, dot, asterisk, semicolon. Line 7: Blank. Line 8: public, class, Bar Chart View Controller, implements, Event Handler, left angle bracket, Action Event, right angle bracket. Line 9: Left brace. Line 10, indented once: private, Bar Chart View, view, semicolon. Line 11, indented once: private, Investment, left square brace, right square brace, investments, semicolon. Line 12: Blank. Line 13, indented once: public, Bar Chart View Controller, left parenthesis, Bar Chart View, view, comma. Line 14, indented a few times: Investment, left square brace, right square brace, investments, right parenthesis. Line 15, indented once: Left brace. Line 16, indented twice: this, dot, investments, equals, investments, semicolon. Line 17, indented twice: this, dot, view, equals, view, semicolon. Line 18, indented twice: left parenthesis, this, dot, view, right parenthesis, dot, set Button Listener, left parenthesis, this, right parenthesis, semicolon. Line 19, indented once: right brace. Line 20: Blank. Line 21, indented once: public, void, handle, left parenthesis, Action Event, event, right parenthesis. Line 22, indented once: left brace. Line 23, indented twice: forward slash, forward slash, update the model. Line 24, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, investments, dot, length, semicolon, i, plus, plus, right parenthesis. Line 25, indented 3 times: investments, left square brace, i, right square, brace, dot, update, left parenthesis, right parenthesis, semicolon. Line 26, indented twice: forward slash, forward slash, update the view. Line 27, indented twice: view, dot, update, left parenthesis, investments, right parenthesis, semicolon. Line 28, indented once: right brace. Line 29: right brace.

Back to Example

Code in the Investment Application dot java file. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, asterisk, Investment Application class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java f x, dot, application, dot, Application, semicolon. Line 6: import, java f x, dot, scene, dot, Scene, semicolon. Line 7: import, java f x, dot, stage, dot, Stage, semicolon. Line 8: Blank. Line 9: public, class, Investment Application, extends, Application. Line 10: Left brace. Line 11, indented once: at, Override. Line 12, indented once: public, void, start, left parenthesis, Stage, stage, right parenthesis. Line 13, indented once: Left brace. Line 14, indented twice: i n t, years, equals, 4, semicolon. Line 15, indented twice: Stocks, stocks, equals, new, Stocks, left parenthesis, years, right parenthesis, semicolon. Line 16, indented twice: Bonds, bonds, equals, new, Bonds, left parenthesis, years, right parenthesis, semicolon. Line 17, indented twice: Investment, left square brace, right square brace, equals, left brace, bonds, comma, stocks, right brace, semicolon. Line 18: Blank. Line 19, indented twice: Bar Chart View, root, equals, new, Bar Chart View, left parenthesis, 2020, comma, investments, right parenthesis, semicolon. Line 20: Blank. Line 21, indented twice: Scene, scene, equals, new, Scene, left parenthesis, root, comma, 540, comma, 510, right parenthesis, semicolon. Line 22, indented twice: stage, dot, set Title, left parenthesis, open quotes, Financial Projections, close quotes, right parenthesis, semicolon. Line 23, indented twice: stage, dot, set Scene, left parenthesis, scene, right parenthesis, semicolon. Line 24, indented twice: stage, dot, show, left parenthesis, right parenthesis, semicolon. Line 25: Blank. Line 26, indented twice: Bar Chart View Controller, controller. Line 27, indented 3 times: equals, new, Bar Chart View Controller, left parenthesis, root, comma, investments, right parenthesis, semicolon. Line 28, indented twice: right brace. Line 29: Blank. Line 30, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 31, indented once: left brace. Line 32, indented twice: launch, left parenthesis, a r g s, right parenthesis, semicolon. Line 33, indented once: right brace. Line 34: right brace.

Back to Example

Code in the Java F X style sheet for the Simple Math application. Program code. In the code, the words in the variable names are merged. Line 1: V Box. Line 2: Left brace. Line 3, indented once: hyphen, f x hyphen background hyphen color, colon, sky blue, semicolon. Line 4: right brace. Line 5: blank. Line 6: Label. Line 7:

Left brace. Line 8, indented once: hyphen f x hyphen background hyphen color: deep sky blue, semicolon. Line 9, indented once: hyphen f x hyphen text hyphen fill, colon, blue, semicolon. Line 10: right brace. Line 11: Blank. Line 12: hash, result. Line 13: Left brace. Line 14, indented once: hyphen f x hyphen background hyphen color, colon, aqua, semicolon. Line 15, indented once: hyphen f x hyphen font hyphen weight, colon, bold, semicolon. Line 16: right brace. Line 17: Blank. Line 18: Button, colon, hover. Line 19: Left brace. Line 20, indented once: hyphen f x hyphen background hyphen color, colon, hash 0 0 F F 0 0, semicolon. Line 21: right brace.

Back to Example

Code in which a style sheet has been imported. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Simple Math Operations Using Buttons. Line 2: Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: Blank. Line 5: import, java, dot, net, dot, U R L, semicolon. Line 6: import, java f x, dot, application, dot, Application, semicolon. Line 7: import, java f x, dot, f x m l, dot, F X M L Loader, semicolon. Line 8: import, java f x, dot, scene, dot, layout, dot, V Box, semicolon. Line 9: import, java f x, dot, scene, dot, Scene, semicolon. Line 10: import, java f x, dot, stage, dot, Stage, semicolon. Line 11: Blank. Line 12: public, class, Simple Math Practice, extends, Application. Line 13: Left brace. Line 14, indented once:

at, Override. Line 15, indented once: public, void, start, left parenthesis, Stage, stage, right parenthesis, forward slash, forward slash, throws Exception. Line 16, indented once: Left brace. Line 17, indented twice: try. Line 18, indented twice: Left brace. Line 19, indented 3 times: U R L, u r l, equals. Line 20, indented 4 times: get Class, left parenthesis, right parenthesis, dot, get Resource, left parenthesis, open quotes, f x m l underscore simple underscore math, dot, f x m l, close quotes, right parenthesis, semicolon. Line 21, indented 3 times: V Box, root, equals, F X M L Loader, dot, load, left parenthesis, u r l, right parenthesis, semicolon. Line 22, indented 3 times: Scene, scene, equals, new, Scene, left parenthesis, root, comma, 300, comma, 275, right parenthesis, semicolon. Line 23, indented 3 times: scene, dot, get Style sheets, left parenthesis, right parenthesis, dot, add, left parenthesis, open quotes, simple underscore math, dot, c s s, close quotes, right parenthesis, semicolon. Line 24, indented 3 times: stage, dot, set Title, left parenthesis, open quotes, Simple Math, close quotes, right parenthesis, semicolon. Line 25, indented 3 times: stage, dot, set Scene, left parenthesis, scene, right parenthesis, semicolon. Line 26, indented 3 times: stage, dot, show, left parenthesis, right parenthesis, semicolon. Line 27, indented twice: right brace. Line 28, indented twice: catch, left parenthesis, Exception, e, right parenthesis. Line 29, indented twice: left brace. Line 30, indented 3 times: e, dot, print Stack Trace, left parenthesis, right parenthesis, semicolon. Line 31, indented twice: right brace. Line 32, indented once: right brace. Line 33: Blank. Line 34, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 35, indented once: left brace. Line 36, indented twice: launch, left parenthesis, a r g s, right parenthesis, semicolon. Line 37, indented once: right brace. Line 38: right brace.

Back to Example

A table detailing the Game View class A P I. Constructor: Game View, left parenthesis, i n t, n Sides, right parenthesis, instantiates a tile puzzle View having an n Sides-by-n Sides grid. Return value: void. Method name and argument list: set Up Game, left parenthesis, i n t, n Sides, right parenthesis, resets the grid as an n Sides-by-n Sides grid.

Back to Table

A table detailing operator precedence. The table has 11 rows and 3 columns. It has the following column headings. Operators. Order of Same-Statement Evaluation. Operation. The row details in the table are the following. Row 1: Operators, Left parenthesis, right parenthesis; Order of Same-Statement Evaluation, left to right; Operation, parentheses for explicit grouping. Row 2: Operators, plus plus, minus minus; Order of Same-Statement Evaluation, right to left; Operation, shortcut post increment and post decrement. Row 3: Operators, plus plus, minus minus, not; Order of Same-Statement Evaluation, right to left; Operation, shortcut preincrement and predecrement, logical unary NOT. Row 4: Operators, asterisk, forward slash, percentage; Order of Same-Statement Evaluation, left or right; Operation, multiplication, division, modulus. Row 5: Operators, plus, minus; Order of Same-Statement Evaluation, left to right; Operation, addition or String concatenation, subtraction. Row 6: Operators, less than, less than or equals, greater than, greater than or equals, instance of; Order of Same-Statement Evaluation, left to right; Operation, relational operators: less than, less than or equal to, greater than, greater than or equal to; instance of. Row 7: Operators, equals equals, not equals; Order of Same-Statement Evaluation, left or right; Operation, equality operators: equal to and not equal to. Row 8: Operators, and and; Order of Same-Statement Evaluation, left tor right; Operation, logical AND. Row 9: Operators, or or; Order of Same-Statement Evaluation, left to right; Operation, logical OR. Row 10: Operators, question mark colon; Order of Same-Statement Evaluation, left to right; Operation, conditional operator. Row 11: Operators, equals, plus equal, minus equals, asterisk equals, forward slash, equals, percentage equals; Order of Same-Statement Evaluation, right to left; Operation, assignment operator and shortcut assignment operators.

Back to Table

The calculation by which the binary number a is converted to its decimal equivalent. a is a positive integer and has the value 0 0 0 0 0 1 0 1 0 1 1 1 1 0 0 1. The value of a is calculated as given below. a equals 2 to the power of 10 plus 2 to the power of 8 plus 2 to the power of 6 plus 2 to the power of 5 plus 2 to the power of 4 plus 2 to the power of 3 plus 2 to the power of 0. This equals 1024 plus 256 plus 64 plus 32 plus 16 plus 8 plus 1. This equals 1401.

Back to Code

The formula to calculate the two's complement of a number. Two's complement of n equals 2 d minus n.

Back to Code

Simplification of the formula to calculate the two's complement of a number. Two's complement of n equals 2 to the power of d minus 1 minus n plus 1.

Back to Code

The calculation for finding out two's complement of b. b is a negative integer that has the value 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0. 2 to the power of d minus 1 is 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1. b is 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0. 2 to the power of d minus 1 minus b is 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1. 2 to the power of d minus 1 minus b plus 1 gives the two's complement of b which is 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0.

Back to Code

Two's complement of b calculated by inverting each bit and adding 1. b is a negative integer that has the value 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0. b inverted is 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1. b inverted plus 1 gives c which is 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0.

Back to Code

Verification that the two's complement of b is correct by calculating the sum of b and c. b is 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0. c is 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0. b plus c is 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.

Back to Code

C is converted to decimal which gives the original number, b. b equals minus of 2 to the power of 5 plus 2 to the power of 2 plus 2 to the power of 1. This equals minus of 32 plus 4 plus 2. This equals minus 38.

Back to Code

The largest positive number that can be represented. 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1. max equals 2 to power of 14 plus 2 to the power of 13 plus 2 to the power of 12 plus 2 to the power of 11 plus 2 to the power of 10 plus 2 to the power of 9 plus 2 to the power of 8 plus 2 to the power of 7 plus 2 to the power of 6 plus 2 to the power of 5 plus 2 to the power of 4 plus 2 to the power of 3 plus 2 to the power of 2 plus 2 to the power of 1 plus 2 to the power of 0.

Back to Code

The smallest negative number that can be represented using 16 bits. 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.

Back to Code

Inverting the bits and adding 1. min is 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0. min inverted is 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1. min inverted plus 1 gives two's complement which is 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.

Back to Code

How single and double precision floating-point numbers are represented. In single-precision floating-point numbers, 1 bit represents the sign, 8 bits represent the biased exponent, and 23 bits represent the mantissa. In double-precision floating-point numbers, 1 bit represents the sign, 11 bits represent the biased exponent, and 52 bits represent the mantissa.

Back to Figure

The bias in floating point numbers is equal to 2 raised to the number of bits of the biased exponent minus 1 the whole minus 1.

Back to Figure

The bias for single precision is 2 raised to the power of 8 minus 1 the whole minus 1 which equals 2 to the power of 7 minus 1 which equals 127.

Back to Figure

How floating-point numbers are represented with real exponents from minus 127 to 128 with a bias of 127. Real exponents are from minus 127 to 0 and to 128. Bias of 127 is added to the real exponents. The biased exponents are 0 to 127 and to 255.

Back to Figure

The bias for double precision. 2 to the power of 11 minus 1 the whole minus 1 which equals 2 to the power of 10 minus 1 which is 1023.

Back to Figure

The form of a floating-point number. (minus 1) to the power of sign times (1 plus significand) times 2 to the power of biased exponent minus bias the whole.

Back to Figure

A single-precision I triple E 754 floating-point number. 0, 1 0 0 0 0 1 1 1, 1 1 0 1 0 0 0 0 and so on 0.

Back to Figure

The biased exponent 1 0 0 0 0 1 1 1 converted to decimal equals 2 to the power of 7 plus 2 to the power of 2 plus 2 to the power of 1 plus 2 to the power of 0. This equals 128 plus 4 plus 2 plus 1. This equals 135.

Back to Figure

Converting a single-precision I triple E 754 floating-point number to decimal. The number equals (minus 1) to the power of 0 times (1 plus .1101) times 2 to the power of 135 minus 127 the whole. This equals 1.1101 times 2 to the power of 8. This equals 1 1 1 0 1 0 0 0 0.

Back to Figure

Converting a single-precision I triple E 754 floating-point number to decimal. In decimal, the number is 2 to the power of 8 plus 2 to the power of 7 plus 2 to the power of 6 plus 2 to the power of 4. This equals 256 plus 128 plus 64 plus 16. This equals 464.

Back to Figure

Alternate method for converting a single-precision I triple E 754 floating-point number to decimal. The calculation equals 1.1101 times 2 to the power of 8. This equals (1 plus 1 times 1 divided by 2 of 1 plus 1 times 1 divided by 2 of 2 plus 0 times 1 divided by 2 of 3 plus 1 times 1 divided by 2 of 4) times 2 to the power of 8. This equals (1 plus 1 divided by 2 plus 1 divided by 4 plus 1 divided by 16) times 2 to the power of 8. This equals (1 plus 1 divided by 2 plus 1 divided by 4 plus 1 divided by 16) times 256. This equals 464.

Back to Figure

y represented in binary. y equals minus 101.11. This equals minus 1.01011 times 2 to the power of 2.

Back to Figure

Biased exponent equals 2 plus 127. This equals 129.

Back to Figure

Converting 129 to binary. 129 equals 1 0 0 0 0 0 0 1.

Back to Figure

The I triple E 754 single-precision value of the number y. 1, 1 0 0 0 0 0 0 1, 0 1 0 1 1 0 0 0 0 and so on to 0.

Back to Figure

The I triple E 754 single-precision value of the number y. 1, 1 0 0 0 0 0 0 1, 0 1 0 1 1 0 0 0 0 and so on to 0.

Back to Table

Line 1: set total to 0.00 dollars. Line 2: reach for first item. Line 3: if item is not the divider bar. Line 4, indented once: add price to total. Line 5: reach for next item. Line 6: if item is not the divider bar. Line 7, indented once: add price to total. Line 7: reach for next item. Line 8: if item is not the divider bar. Line 8, indented once: add price to total. Line 9: dot dot dot, left parentheses, finally, right parentheses. Line 10: reach for next item. Line 11: item is the divider bar, comma. Line 11, indented once: tell teh customer the total price.

Back to Code

Program code. The code contains the keyword while. Line 1: 2 forward slashes. initialize variables. Line 2: while, left parentheses, condition, right parentheses. Line 3: left brace. Line 4, indented once: 2 forward slashes. process data, semicolon, loop body. Line 5: right brace. Line 6: 2 forward slashes. process the results.

Back to Code

Line 1: set total to 0.00 dollars. Line 2: reach for first item. Line 3: while item is not the divider bar. Line 4: left brace. Line 5, indented once: add price to total. Line 6, indented once: reach for next item. Line 7: right brace. Line 8: 2 forward slashes. if we get here, comma, the item is the divider bar output the total price.

Back to Code

while, left parentheses, condition, right parentheses, semicolon. 2 forward slashes. semicolon causes endless loop if condition is true.

Back to Code

Line 1: initialize variables. Line 2: read the first data item.

2 forward slashes. priming read. Line 3: while data item is not the sentinel value. Line 4: left brace. Line 5, indented once: process the data. Line 6: blank. Line 7, indented once: read the next data item. 2 forward slashes. update read. Line 8: right brace. Line 9: report the results.

Back to Code

Line 1: initialize variables. Line 2: while data item is not the sentinel value. Line 3: left brace. Line 4, indented once: read the next data. Line 5, indented once: process the data. Line 6: right brace. Line 7: report the results.

Back to Code

Line 1: initialize variables. Line 2: while we have not reached end of file. Line 3: left brace. Line 4, indented once: read the next data item. Line 5, indented once: process the data. Line 6: right brace. Line 7: report the results.

Back to Code

Line 1: set total to 0. Line 2: read a number, 2 forward slashes, priming read. Line 3: while the number is not the sentinel value. Line 4: left brace. Line 5, indented once: add the number to total. Line 6, indented once: read the next number, 2 forward slashes, update read. Line 7: right brace. Line 8: output the total.

Back to Code

Line 1: set count Passed to 0. Line 2: read a test score. Line 3: while the test score is not the sentinel value. Line 4: left brace. Line 5, indented once: if the test score, right angled bracket, equals sign, 60.

Back to Code

Line 1: set total to 0. Line 2: set count to 0. Line 3: read a number. Line 4: while the number is not the sentinel value. Line 5: left brace. Line 6, indented once: add the number to total. Line 7, indented once: add 1 to the count. Line 8: blank. Line 9, indented once: read the next number. Line 10: right brace. Line 11: right brace. Line 12: set the average to total, forward slash, count. Line 13: output the average.

Back to Code

double average equals left parentheses, double, right parentheses, total, forward slash, count, semicolon.

Back to Code

Line 1: read a first number and make it the maximum. Line 2: while there is another number to read. Line 3: left brace. Line 4, indented once: read the next number. Line 5, indented once: if number greater than maximum. Line 6, indented once: left brace. Line 7, indented twice: set maximum to number. Line 8, indented once: right brace. Line 8: right brace. Line 9: output the maximum.

Back to Code

while, left parentheses, option exclamation mark, equals sign, single quote, uppercase s, single quote, 2 pipes, option, exclamation mark, equals sign, single quote, lowercase s, single quote, right parentheses, 2 forward slashes, INCORRECT.

Back to Code

left parentheses, option, 2 equals sign, single quote, uppercase s, single quote, 2 pipes, option, 2 equals sign, single quote, lowercase s, single quote, right parentheses.

Back to Code

exclamation mark, left parentheses, option, 2 equals sign, single quote, uppercase s, single quote, 2 pipes, option, 2 equals sign, single quote, lowercase s, single quote, right parentheses.

Back to Code

left parentheses, option, exclamation mark, equals sign, single quote, uppercase s, single quote, 2 ampersands, option, exclamation mark, single quote, lowercase s, single quote, right parentheses.

Back to Code

Line 1: 2 forward slashes. initialize variables. Line 2: do. Line 3: left brace. Line 4, indented once: 2 forward slashes. body of loop. Line 5: right brace, while, left parentheses, condition, right parentheses, semicolon. Line 6: 2 forward slashes. process the results.

Back to Code

Program code. The code contains the keyword if. Line 1: System dot out dot print, left parentheses, double quotes, Enter a number between 1 and 10, right angled bracket, double quotes, right parentheses, semicolon. Line 2: number equals scan dot next Int, left parentheses, right parentheses, semicolon. Line 3: if, left parentheses, number less than 1, 2 pipes, number greater than 10, right parentheses, 2 forward slashes, INCORRECT, exclamation mark. Line 4: left brace. Line 5, indented once: System dot out dot print, left parentheses, double quotes, Enter a number between 1 and 10, right angled bracket, double quotes, right parentheses, semicolon. Line 6, indented once: number equals scan dot next Int, left parentheses, right parentheses, semicolon. Line 7: right brace.

Back to Code

Item get Next, left parentheses, right parentheses.

Back to Code

Line 1: Item new Item, semicolon. Line 2: new Item, equals, get Next, left parentheses, right parentheses, semicolon.

Back to Code

double get Price, left parentheses, right parentheses.

Back to Code

Line 1: Item new Item, semicolon. Line 2: double price, semicolon. Line 3: blank. Line 4: new Item equals get Next, left parentheses, right parentheses, semicolon. Line 5: price equals new Item dot get Price, left parentheses, right parentheses, semicolon.

Back to Code

void animate, left parentheses, double sub total, right parentheses.

Back to Code

Line 1: for, left parentheses, initialization, semicolon, loop condition, semicolon, loop update, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes. loop body. Line 4: right brace.

Back to Code

Line 1: 2 forward slashes. initialization. Line 2: while, left parentheses, loop condition, right parentheses. Line 3: left brace. Line 4, indented once: 2 forward slashes, loop body. Line 5, indented once: 2 forward slashes, loop update. Line 6: right brace.

Back to Code

Line 1: set total to 0. Line 2: for i equals 1 to 5 by 1. Line 3: left brace. Line 4, indented once: read integer. Line 5, indented once: add integer to total. Line 6: right brace. Line 7: print the total.

Back to Code

Line 16: for, left parentheses, int i equals 1, semicolon, i less than or equal to 5, semicolon, i plus plus, right parentheses.
Line 17: left brace.
Line 18, indented once: System dot out dot print, left parentheses, double quotes, Enter an integer, right angled bracket, double quotes, right parentheses, semicolon.
Line 19, indented once: number equals scan dot next Int, left parentheses, right parentheses, semicolon.
Line 20: blank.
Line 21, indented once: total plus equals number, semicolon. 2 forward slashes, add input to total.
Line 22: right brace.
Line 23: blank.
Line 24: System dot out dot print l n, left parentheses, double quotes, The total for the, double quotes, plus, left parentheses, i minus 1, right parentheses.
Line 25, indented multiple times: plus, double quotes, numbers is, double quotes, plus, total, right parentheses, semicolon, 2 forward slashes, i is out of scope.

Back to Code

Line 16: for, left parentheses, int i equals 1, semicolon, i less than or equal to 5, semicolon, i plus plus, right parentheses.
Line 17: left brace.
Line 18, indented once: System dot out dot print, left parentheses, double quotes, Enter an integer, right angled bracket, double quotes, right parentheses, semicolon.
Line 19, indented once: number equals scan dot next Int, left parentheses, right parentheses, semicolon.
Line 20: blank.
Line 21, indented once: total plus equals number, semicolon, 2 forward slashes, add input to total.
Line 22: right brace.
Line 23: blank.
Line 24: for, left parentheses, int i equals 1, semicolon, i less than or equal to 10, semicolon, i plus plus, right parentheses.
Line 25: left brace.
Line 26, indented once: System dot out dot print, left parentheses, double quotes, Enter integer, double quotes, plus i plus, double quotes, right angled bracket, double quotes, right parentheses, semicolon.
Line 27, indented once: number equals scan dot next Int, left parentheses, right parentheses, semicolon.
Line 28: right brace.

Back to Code

Line 15: int i, semicolon.
Line 16: for, left parentheses, i equals 1, semicolon, i less than or equal to 5, semicolon, i plus plus, right parentheses.
Line 17: left brace.
Line 18, indented once: System dot out dot print, left parentheses, double quotes, Enter an integer, right angled bracket, double quotes, right parentheses, semicolon.
Line 19, indented once: number equals scan dot next Int, left parentheses, right parentheses, semicolon.
Line 20: blank.
Line 21, indented once: total plus equals number, semicolon. 2 forward slashes, add input to total.
Line 22: right brace.
Line 23: blank.
Line 24: System dot out dot print l n, left parentheses, double quotes, The total for, double quotes, plus, left parentheses, i minus 1, right parentheses, 2 forward slashes, i is 6.
Line 25, indented multiple times: plus, double quotes, numbers is, double quotes, plus, total, right parentheses, semicolon.

Back to Code

Line 1: set output to an empty string. Line 2: for i equals 0 to 20 by 2. Line 3: left brace. Line 4, indented once: append i and a space to the output string. Line 5: right brace. Line 6: print the output String.

Back to Code

Line 1: read a sentence. Line 2: remove leading and trailing white space. Line 3: if sentence is empty. Line 4, indented once: print message no tokens found. Line 5: else. Line 6, indented once: set count Tokens equals 1. Line 7, indented once: initialize flag variable to false to indicate that the current character is not. Line 8, indented once: white space. Line 9, indented once: for i equals 0 to left parentheses, sentence length minus 1, right parentheses, by 1. Line 10, indented once: left brace. Line 11, indented twice: get character at position i in sentence. Line 12, indented twice: if, left parentheses, character is a space or tab, right parentheses. Line 13, indented thrice: if flag variable is false. Line 14, indented 4 times: we have found the end of a token, comma, so increment count Tokens. Line 15, indented 4 times: set flag variable to true to indicate that current character is. Line 16, indented 4 times: white space. Line 17, indented twice: else. Line 18, indented thrice: set flag variable to false to indicate that current character is not. Line 19, indented thrice: white space. Line 20, indented once: right brace. Line 21, indented once: print the number of tokens found.

Back to Code

Line 1: set backwards to an empty String. Line 2: read a sentence. Line 3: blank. Line 4: for i equals left parentheses, length of sentence minus 1, right parentheses, to 0 by negative 1. Line 5: left brace. Line 6, indented once: get character at position i in sentence. Line 7, indented once: append character to backwards. Line 8: right brace. Line 9: print backwards.

Back to Code

Line 1: for diameter equals 300 to 25 by negative 25. Line 2: left brace. Line 3, indented once: draw the circle with current diameter. Line 4, indented once: if color is black. Line 5, indented twice: set color to red. Line 6, indented once: else. Line 7, indented twice: set color to black. Line 8: right brace.

Back to Code

Line 1: g c dot fill Oval, left parentheses, center uppercase x, minus, diameter, forward slash 2, comma. Line 2, indented multiple times: center Y minus diameter forward slash 2, comma. Line 3, indented multiple times: diameter, comma, diameter, right parentheses, semicolon.

Back to Code

for, left parentheses, int i equals 0, semicolon, i less than 5, semicolon, i plus plus, right parentheses, 2 forward slashes, executes 5 times.

Back to Code

for, left parentheses, int i equals 1, semicolon, i less than or equal to 5, semicolon, i plus plus, right parentheses, 2 forward slashes, executes 5 times.

Back to Code

for, left parentheses, int i equals 1, semicolon, i less than 5, semicolon, i plus plus, right parentheses, 2 forward slashes, INCORRECT, exclamation mark, executes only 4 times.

Back to Code

int i equals original dot length, left parentheses, right parentheses, minus 1, semicolon.

Back to Code

System dot out dot print l n, left parentheses, double quotes, The sentence backwards is, colon, back slash n, double quotes, plus, backwards, right parentheses, semicolon.

Back to Code

Line 1: look for a customer. Line 2: while there is a customer in line. Line 3: left brace. Line 4, indented once: set total to dollar, 0.00. Line 5, indented once: reach for first item. Line 6, indented once: while item is not the divider bar. Line 7, indented once: left brace. Line 8, indented twice: add price to total. Line 9, indented twice: reach for next item. Line 10, indented once: right brace. Line 11, indented once: 2 forward slashes, if we get here, comma, the item is the divider bar. Line 12, indented once: output the total price. Line 13: blank. Line 14, indented once: look for another customer. Line 15: right brace.

Back to Code

The printed numbers are as follows. Row 1: 1. Row 2: 1 2. Row 3: 1 2 3. Row 4: 1 2 3 4. Row 5: 1 2 3 4 5.

Back to Code

Line 1: for line equals 1 to 5 by 1. Line 2: left brace. Line 3, indented once: for number equals 1 to line by 1. Line 4, indented once: left brace. Line 5, indented twice: print number and a space. Line 6, indented once: right brace. Line 7, indented once: print a new line. Line 8: right brace.

Back to Code

Line 1: read first number. 2 forward slashes, priming read. Line 2: while number is not 0. Line 3: left brace. Line 4: blank. Line 5, indented once: print, double quotes, The factors for number are, blank space, double quotes. Line 6, indented once: for factor equals 2 to left parentheses, number, forward slash 2, right parentheses, by 1. Line 7, indented once: left brace. Line 8, indented twice: if number modulus factor is 0. Line 9, indented thrice: print factor and a space. Line 10, indented once: right brace. Line 11, indented once: print a new line. Line 12: blank. Line 13, indented once: read next number. 2 forward slashes, update read. Line 14: right brace.

Back to Code

Line 1: read first number. 2 forward slashes, priming read. Line 2: while number is not 0. Line 3: left brace. Line 4, indented once: print, double quotes, The factors for number are, blank space, double quotes. Line 5, indented once: set flag to false. Line 6, indented once: for factor equals 2 to, left parentheses, number, forward slash, 2, right parentheses, by 1. Line 7, indented once: left brace. Line 8, indented twice: if number percentage factor is 0. Line 9, indented twice: left brace. Line 10, indented thrice: print factor and a space. Line 11, indented thrice: set flag to true. Line 12, indented twice: right brace. Line 13, indented once: right brace. Line 14, indented once: if flag is false. Line 15, indented twice: print, double quotes, number is prime, double quotes. Line 16: blank. Line 17, indented once: print a new line. Line 18: blank. Line 19, indented once: read next number, 2 forward slashes, update read. Line 20: right brace.

Back to Code

Line 1: Item new Item, semicolon. Line 2: new Item, equals, get Next, left parentheses, right parentheses, semicolon.

Back to Code

double get Price, left parentheses, right parentheses.

Back to Code

Line 1: Item new Item, semicolon. Line 2: double price, semicolon. Line 3: new Item, equals, get Next, left parentheses, right parentheses, semicolon. Line 4: price equals new Item dot get Price, left parentheses, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: int, while. Line 1: System dot out dot print l n, left parentheses, double quotes, Enter an int, right angled bracket, double quotes, right parentheses, semicolon. Line 2: int i equals scan dot next Int, left parentheses, right parentheses, semicolon. Line 3: while, left parentheses, i, exclamation mark, equals sign, negative 1, right parentheses. Line 4: left brace. Line 5, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 6: blank. Line 7, indented once: System dot out dot print l n, left parentheses, double quotes, Enter an int, right angled bracket, double quotes, right parentheses, semicolon. Line 8, indented once: i equals scan dot next Int, left parentheses, right parentheses, semicolon. Line 9: right brace.

Back to Code

Program code. The code contains the following keywords: int, while. Line 1: int i equals 0, semicolon. Line 2: while, left parentheses, i exclamation mark, equals sign, negative 1, right parentheses. Line 3: left brace. Line 4, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 5, indented once: System dot out dot print, left parentheses, double quotes, Enter an int, right angled bracket, double quotes, right parentheses, semicolon. Line 6, indented once: i equals scan dot next Int, left parentheses, right parentheses, semicolon. Line 7: right brace.

Back to Code

Program code. The code contains the following keywords: int, while. Line 1: System dot out dot print l n, left parentheses, double quotes, Enter an int, right angled bracket, double quotes, right parentheses, semicolon. Line 2: int i equals scan dot next Int, left parentheses, right parentheses, semicolon. Line 3: while, left parentheses, i exclamation mark, equals sign, negative 1, right parentheses. Line 4: left brace. Line 5, indented once: System dot out dot print l n, left parentheses, double quotes, Enter an int, right angled bracket, double quotes, right parentheses, semicolon. Line 6, indented once: i equals scan dot next Int, left parentheses, right parentheses, semicolon. Line 7: blank. Line 8, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 9: right brace.

Back to Code

Program code. The code contains the following keywords: int, while. Line 1: int sum equals 0, semicolon. Line 2: int i equals 17, semicolon. Line 3: while, left parentheses, i percentage sign 10, exclamation mark, equals sign, 0, right parentheses. Line 4: left brace. Line 5, indented once: sum, plus, equals, i, semicolon. Line 6, indented once: i plus, plus, semicolon. Line 7: right brace.

Back to Code

Program code. The code contains the following keywords: int, do, while. Line 1: int i equals 6, semicolon. Line 2: int product equals 1, semicolon. Line 3: do. Line 4: left brace. Line 5, indented once: product, asterisk equals, i, semicolon. Line 6, indented once: i plus, plus, semicolon. Line 7: right brace, while, left parentheses, i less than 9, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: int, do, while. Line 1: int i equals 6, semicolon. Line 2: int product equals 1, semicolon. Line 3: do. Line 4: left brace. Line 5, indented once: product, asterisk equals, i, semicolon. Line 6, indented once: i plus, plus. Line 7: right brace, while, left parentheses, product less than 9, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords:

for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i less than 3, semicolon, i plus plus, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 3: System dot out dot print l n, left parentheses, double quotes, Done, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i less than or equal to 2, semicolon, i plus plus, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 3: System dot out dot print l n, left parentheses, double quotes, Done, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: int, for. Line 1: int i equals 0, semicolon. Line 2: for, left parentheses, i equals 0, semicolon, i less than or equal to 2, semicolon, i plus plus, right parentheses. Line 3, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: int, for. Line 1: int i equals 0, semicolon. Line 2: for, left parentheses, i equals 0, semicolon, i less than 2034, semicolon, i plus, plus, right parentheses. Line 3, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the keywords: int, for. Line 1: int i equals 0, semicolon. Line 2: int sum equals 0, semicolon. Line 3: for, left parentheses, i equals 0, semicolon, i less than 5, semicolon, i plus, plus, right parentheses. Line 4: left brace. Line 5, indented once: sum plus equals i, semicolon. Line 6: right brace.

Back to Code

Program code. The code contains the following keywords: int, for, if. Line 1: int i equals 0, semicolon. Line 2: int sum equals 0, semicolon. Line 3: for, left parentheses, i equals 0, semicolon, i less than 40, semicolon, i plus, plus, right parentheses. Line 4: left brace. Line 5, indented once: if, left parentheses, i, percentage sign, 10, equals, equals, 0, right parentheses. Line 6, indented twice: sum, plus, equals, i, semicolon. Line 7: right brace.

Back to Code

Program code. The code contains the following keywords:

int, for. Line 1: int sum equals 0, semicolon. Line 2: for, left parentheses, int i equals 1, semicolon, i less than 10, semicolon, i plus, plus, right parentheses. Line 3: left brace. Line 4, indented once: i plus, plus, semicolon. Line 5, indented once: sum, plus, equals, i, semicolon. Line 6: right brace.

Back to Code

Program code. The code contains the keywords: int, for. Line 1: int sum equals 0, semicolon. Line 2: for, left parentheses, int i equals 10, semicolon, i greater than 5, semicolon, i minus, minus, right parentheses. Line 3: left brace. Line 4, indented once: sum, plus, equals, i, semicolon. Line 5: right brace.

Back to Code

Program code. The code contains the following keywords: for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i less than 5, semicolon, i plus, plus, right parentheses. Line 2: left brace. Line 3, indented once: System dot out dot print l n, left parentheses, Math dot max, left parentheses, i, comma, 3, right parentheses, right parentheses, semicolon. Line 4: right brace.

Back to Code

Program code. The code contains the following keywords: int, while. Line 1: int i equals 0, semicolon. Line 2: int sum equals 0, semicolon. Line 3: while, left parentheses, i, exclamation mark, equals sign, 7, right parentheses. Line 4: left brace. Line 5, indented once: sum, plus, equals, i, semicolon. Line 6, indented once: i plus, plus, semicolon. Line 7: right brace.

Back to Code

Line 1: Random random equals new Random, left parentheses, right parentheses, semicolon. Line 2: int i equals random dot next Int, left parentheses, 5, right parentheses, plus, 3, semicolon.

Back to Code

Line 1: System dot out dot print, left parentheses, double quotes, Enter a starting integer, right angled bracket, double quotes, right parentheses, semicolon. Line 2: int start, equals, scan dot next Int, left parentheses, semicolon. Line 3: 2 forward slashes, your code goes here.

Back to Code

Line 1: System dot out dot print, left parentheses, double quotes, Enter an integer value, comma, double quotes. Line 2, indented multiple times: plus, double quotes, enter negative 1 to stop, right angled bracket, double quotes, right parentheses, semicolon. Line 3: int value equals scan dot next Int, left parentheses, right parentheses, semicolon. Line 4: 2 forward slashes, your code goes here.

Back to Code

Line 1: int sum equals 0, semicolon. Line 2: int count Multiples Of 7 equals 0, semicolon. Line 3: int count equals 1, semicolon. Line 4:2 forward slashes, your code goes here.

Back to Code

Program code. The code contains the following keywords: String, while. Line 1: String sentence equals, double quotes, double quotes, semicolon. Line 2: String word, semicolon. Line 3: 2 forward slashes, your code goes here. Line 4: blank. Line 5: while, left parentheses, exclamation mark, word dot equals, left parentheses, double quotes, end, double quotes, right parentheses, right parentheses. Line 6: left brace. Line 7, indented once: 2 forward slashes, and your code goes here. Line 8: blank. Line 9: right brace. Line 10: System dot out dot print l n, left parentheses, double quotes, The sentence is, double quotes, plus, sentence, right parentheses, semicolon.

Back to Code

Line 1: int sum equals 0, semicolon. Line 2: 2 forward slashes, your code goes here.

Back to Code

Program code. The code contains the following keywords:

int, while. Line 1: int i equals 0, semicolon. Line 2: while, left parentheses, i less than 5, right parentheses. Line 3: left brace. Line 4, indented once: System dot out dot print l n, left parentheses, double quotes, Hi there, double quotes, right parentheses, semicolon. Line 5, indented once: i plus, plus. Line 6: right brace. Line 7: blank. Line 8: 2 forward slashes, your code goes here.

Back to Code

Line 1: int sum equals 0, semicolon. Line 2: 2 forward slashes, your code goes here.

Back to Code

Program code. The code contains the keywords: int, while. Line 1: int i equals 0, semicolon. Line 2: while, left parentheses, i less than 3, right parentheses. Line 3, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: int, while. Line 1: int num, semicolon. Line 2: Scanner scan equals new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 3: while, left parentheses, num, exclamation mark, equals sign, negative 1, right parentheses. Line 4: left brace. Line 5, indented once: System dot out dot print, left parentheses, double quotes, Enter an integer, right angled bracket, double quotes, right parentheses, semicolon. Line 6, indented once: num equals scan dot next Int, left parentheses, right parentheses, semicolon. Line 2, indented once: System dot out dot print l n, left parentheses, num, right parentheses, semicolon. Line 3: right brace.

Back to Code

Program code. The code contains the keyword for. Line 1: for, left parentheses, int i equals 0, semicolon, i less than 3, semicolon, i plus, plus, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the keyword for. Line 1: for, left parentheses, int i equals 10, semicolon, i greater than 0, semicolon, i plus, plus, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: int, while. Line 1: Random random equals new Random, left parentheses, right parentheses, semicolon. Line 2: int number equals 1 plus random dot next Int, left parentheses, 10, right parentheses, semicolon. Line 3: while, left parentheses, number, exclamation mark, equals, 5, 2 pipes, number, exclamation mark, equals, 7, right parentheses. Line 4: left brace. Line 5, indented once: number equals 1 plus random dot next Int, left parentheses, 10, right parentheses, semicolon. Line 6: right brace. Line 7: System dot out dot print l n, left parentheses, double quotes, The number is, double quotes, plus, number, right parentheses, semicolon.

Back to Code

Program code. The code contains the keywords: int, for. Line 1: int sum equals 0, semicolon. Line 2: for, left parentheses, int i equals 1, semicolon, i less than 6, semicolon, i plus, plus, right parentheses. Line 3, indented once: sum plus, equals, i, semicolon. Line 4: blank. Line 5: System dot out dot print l n, left parentheses, double quotes, The value of i is, double quotes, plus i, right parentheses, semicolon.

Back to Code

Program code. The code contains the keywords int, dot, while. Line 1: int i equals 0, semicolon. Line 2: int sum equals 0, semicolon. Line 3: do. Line 4: left brace. Line 5, indented once: sum plus equals i, semicolon. Line 6, indented once: i plus, plus, semicolon. Line 7: right brace, while, left parentheses, i less than 3, right parentheses, 2 forward slashes, line 11.

Back to Code

Line 1: Test dot java, colon, 11, colon, error, colon, single quote, semicolon, single quote, expected. Line 2: while, left parentheses, i less than 3, right parentheses, 2 forward slashes, line 11. Line 3: a caret sign below the first forward slash in Line 2. Line 4: 1 error.

Back to Code

Program code. The code contains the keywords: int, while. Line 1: int i equals 0, semicolon. Line 2: while, left parentheses, i less than 3, right parentheses. Line 3: left brace. Line 4, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 5, indented once: i minus, minus, semicolon. Line 6: right brace.

Back to Code

Program code. The code contains the keywords: for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i plus, plus, semicolon, i less than 3, right parentheses, 2 forward slashes, line 5. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon.

Back to Code

Line 1: Test dot java, colon, 5, colon, error, colon, not a statement. Line 2: for, left parentheses, int i equals 0, semicolon, i plus, plus, semicolon, i less than 3, right parentheses, 2 forward slashes, line 5. Line 3: a caret sign under the less than sign in Line 2. Line 4: 1 error.

Back to Code

Program code. The code contains the keywords: for, int. Line 1: for, left parentheses, int i equals 1, semicolon, i less than 3, semicolon, i plus plus, right parentheses, 2 forward slashes, line 5. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the keywords: int, for. Line 1: int product equals 1, semicolon. Line 2: for, left parentheses, int i equals 1, comma, i less than 5, comma, i plus, plus, right parentheses, 2 forward slashes, line 8. Line 3, indented once: product, asterisk, equals, i, semicolon. Line 4: System dot out dot print l n, left parentheses, double quotes, Product is, double quotes, plus, product, right parentheses, semicolon. 2 forward slashes, line 10.

Back to Code

Line 1: Test dot java, colon, 8, colon, error, colon, single quote, semicolon, single quote, expected. Line 2, indented once: for, left parentheses, int i equals 1, comma, i less than 5, comma, i plus, plus, right parentheses, 2 forward slashes, line 8. Line 3: a caret sign below the less than symbol in Line 2. Line 4: Test dot java, colon, 8, colon, error, colon, illegal start of type. Line 5, indented once: for, left parentheses, int i equals 1, comma, i less than 5, comma, i plus, plus, right parentheses, 2 forward slashes, line 8. Line 6: a caret sign under 5 in Line 5. Line 7: 2 errors.

Back to Code

Program code. The code contains the following keywords: for, int. Line 1: for, left parentheses, int i equals 0, semicolon, i less than 3, semicolon, i plus, plus, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 3: System dot out dot print l n, left parentheses, double quotes, i equals, double quotes, plus, i, right parentheses, semicolon, 2 forward slashes, line 8.

Back to Code

Line 1: Test dot java, colon, 8, colon, error, colon, cannot find symbol. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, i equals, double quotes, plus, i, right parentheses, semicolon, 2 forward slashes, line 8. Line 3: a caret sign under the second lowercase i in Line 2. Line 4: symbol, colon, variable i. Line 5: location, colon, class Test. Line 6: 1 error.

Back to Code

Program code. The code contains the following keywords: int, for. Line 1: int i equals 0, semicolon. Line 2: for, left parentheses, int i equals 0, semicolon, i less than 3, semicolon, i plus, plus, right parentheses, 2 forward slashes, line 6. Line 3, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon.

Back to Code

Line 1: Test dot java, colon, 6, colon, error, colon, variable i is already defined in method main, left parentheses, string, left square bracket, right square bracket, right parentheses. Line 2: for, left parentheses, int i equals 0, semicolon, i less than 3, semicolon, i plus, plus, right parentheses, 2 forward slashes, line 6. Line 3: a caret sign under the first lowercase i in Line 2. Line 3: 1 error.

Back to Code

The syntax is as follows. Line 1: access Modifier class Class Name. Line 2: left brace. Line 3, indented once: 2 forward slashes, class definition goes here. Line 4: right brace.

Back to Code

Line 1: public class Auto. Line 2: left brace. Line 3: right brace.

Back to Code

The syntax is as follows. access Modifier data Type identifier List, semicolon.

Back to Code

Line 1: private String name equals, double quotes, double quotes, semicolon, 2 forward slashes, an empty string. Line 2: private final int PERFECT underscore SCORE equals 100, comma, PASSING underscore SCORE equals 60, semicolon. Line 3: private int start X, comma, start Y, comma, width, comma, height, semicolon.

Back to Code

Program code. The code contains the keywords public, class, private, String, int, and double. Line 1: public class Auto. Line 2: left brace. Line 3, indented once: private String model, semicolon. Line 4, indented once: private int miles Driven, semicolon. Line 5, indented once: private double gallons Of Gas, semicolon. Line 6: right brace.

Back to Code

The syntax is as follows. Line 1: access Modifier return Type method Name, left parentheses, parameter list, right parentheses, 2 forward slashes, method header. Line 2: left brace. Line 3, indented once: 2 forward slashes, method body. Line 4: right brace.

Back to Code

The syntax is as follows. illegal start of expression.

Back to Code

The syntax is as follows. single quote, semicolon, single quote, expected.

Back to Code

Program code. The code contains the keywords public, static, void, and String. Line 1: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, application code. Line 4: right brace.

Back to Code

The syntax is as follows. return expression, semicolon.

Back to Code

The syntax is as follows. return, semicolon.

Back to Code

The syntax is as follows. Line 1: access Modifier Class Name, left parentheses, parameter list, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, constructor body. Line 4: right brace.

Back to Code

Program code. The code contains the keywords public, void, String, int, double. Line 1: 2 forward slashes, Error, exclamation mark, void return value specified. Line 2: public void Auto, left parentheses, String model, comma. Line 3, indented multiple times: int start Miles Driven, comma. Line 4, indented multiple times: double start Gallons Of Gas, right parentheses. Line 5: left brace. Line 6, indented once: 2 forward slashes, body of constructor. Line 7: right brace.

Back to Code

Auto g m, equals, new Auto, left parentheses, double quotes, Prius, double quotes, comma, 350, comma, 15.5, right parentheses, semicolon.

Back to Code

Line 1: Auto Client, dot, java, colon, 13, colon, error, colon, constructor Auto in class Auto cannot be applied to given types, semicolon. Line 2, indented once: Auto g m, equals, new Auto, left parentheses, double quotes, Prius, double quotes, comma, 350, comma, 15.5, right parentheses, semicolon. Line 3: a caret sign under n of new in Line 2. Line 4, indented once: required, colon, no arguments. Line 5, indented once: found, colon, String, comma, int, comma, double. Line 6, indented once: reason, colon, actual and formal argument lists differ in length. Line 7: 1 error.

Back to Code

cannot find symbol.

Back to Code

Program code. The code contains the keywords public and void. Line 1: pubilc void set Instance Variable, left parentheses, data Type new Value, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, validate new Value, comma, then assign to the instance variable. Line 4: right brace.

Back to Code

Program code. The code contains the keywords public, void and String. Line 1: 2 forward slashes, Incorrect, exclamation mark, parameter hides instance variable. Line 2: public void set Model, left parentheses, String model, right parentheses. Line 3: left brace. Line 4, indented once: model equals model, semicolon. Line 5: right brace.

Back to Code

Program code. The code contains the keywords public, void, and String. Line 1: 2 forward slashes, Incorrect, exclamation mark, declared local variable hides instance variable. Line 2: public void set Model, left parentheses, String new Model, right parentheses. Line 3: left brace. Line 4, indented once: String model, semicolon, 2 forward slashes, declared variable hides instance variable. Line 5, indented once: model equals new Model, semicolon. Line

6: right brace.

Back to Code

Program code. The code contains the keywords public, void, and String. Line 1: 2 forward slashes, Incorrect, exclamation mark, Declaring the parameter, semicolon, parameters are declared already. Line 2: public void set Model, left parentheses, String new Model, right parentheses. Line 3: left brace. Line 4, indented once: String new Model, semicolon. 2 forward slashes, local variable has same name as parameter. Line 5, indented once: model equals new Model, semicolon. Line 6: right brace.

Back to Code

variable new Model is already defined in method set Model, left parentheses, String, right parentheses.

Back to Code

Program code. The code contains the keywords public, void, String, and this. Line 1: public void set Model, left parentheses, String model, right parentheses. Line 2: left brace. Line 3, indented once: this dot model equals model, semicolon. Line 4: right brace.

Back to Code

Program code. The code contains the keywords public, String, this, and return. Line 1: public Auto set Model, left parentheses, String model, right parentheses. Line 2: left brace. Line 3, indented once: this dot model equals model, semicolon. Line 4, indented once: return this, semicolon. Line 5: right brace.

Back to Code

sporty dot set Gallons Of Gas, left parentheses, 3.4, right parentheses, dot, set Miles Driven, left parentheses, 67, right parentheses, semicolon.

Back to Code

Left parentheses, sporty dot set Gallons Of Gas, left parentheses, 3.4, right parentheses, right parentheses, dot set Miles Driven, 67, right parentheses, semicolon.

Back to Code

Left parentheses, sporty, right parentheses, dot, set Miles Driven, left parentheses, 67, right parentheses, semicolon.

Back to Code

Line 1: public String to String, left parentheses, right parentheses. Line 2: public boolean equals, left parentheses, Object o, right parentheses.

Back to Code

Program code. The code contains the keyword if. Line 1:

if, left parentheses, o, instance Of, Auto, right parentheses. Line 2: left brace. Line 3, indented once: Auto o b j Auto equals, left parentheses, Auto, right parentheses, o, semicolon. Line 3, indented once: 2 forward slashes, call methods of the Auto class with o b j Auto.

Back to Code

The error reads: non, hyphen, static variable x x x cannot be referenced from a static context.

Back to Code

a forward slash b, plus, c forward slash d, equals, left parentheses, a asterisk d, right parentheses, forward slash, left parentheses, b asterisk d, right parentheses, plus, left parentheses, b asterisk c, right parentheses, forward slash, left parentheses, b asterisk d, right parentheses, equals, left parentheses, a asterisk d, plus, b asterisk c, right parentheses, forward slash, left parentheses, b asterisk d, right parentheses.

Back to Code

left parentheses, a, forward slash, b, right parentheses, asterisk, left parentheses, c, forward slash, d, right parentheses, equals, left parentheses, a asterisk c, right parentheses, forward slash, left parentheses, b asterisk d, right parentheses.

Back to Code

Line 1: a equals g asterisk a 1. Line 2: b equals g asterisk b 1.

Back to Code

The steps are as follows. Step 1. 1635 equals 75 asterisk 21 plus 60, left parentheses, a equals 1635, comma, b equals 75, comma, q equals, 21, comma, r equals 60, right parentheses. Line 2: 75 equals 60 asterisk 1 plus 15, left parentheses, a equals 75, comma, b equals 60, comma, q equals 1, comma, r equals 15, right parentheses. Line 3: 60 equals 15 asterisk 4 plus 0, left parentheses a equals 60, comma, b equals 15, comma, q equals 4, comma, r equals 0, right parentheses.

Back to Code

The output reads: 10, forward slash, 5, in lowest terms is equal to 2.

Back to Code

Rational r 3, equals, r 1 dot add, left parentheses, r 2, right parentheses, semicolon. 2 forward slashes, r 1 and r 2 are two existing Rationals.

Back to Code

The output reads: 3, forward slash, 5, plus, 2, forward slash, 3, equals, 19, forward slash, 15.

Back to Code

The syntax is as follows. enum Enum Name, left brace, o b j 1, comma, o b j 2, comma, dot, dot, dot, right brace, semicolon, where o b j 1, o b j 2, et cetera, are names for the constant objects.

Back to Code

enum Days, left parentheses, SUN, comma, MON, comma, T U E, comma, WED, comma, T H U R, comma, FRI, comma, SAT, right brace, semicolon. The names of the constant objects are in uppercase.

Back to Code

The syntax is as follows: enum Type dot enum Object.

Back to Code

Days dot WED.

Back to Code

The error reads: an enum switch case label must be the unqualified name of an enumeration constant.

Back to Code

if, left parentheses, airport 1, 2 equals sign, airport 2, right parentheses.

Back to Code

The command is as follows: java doc Auto dot java.

Back to Code

The command is as follows: java doc asterisk dot java.

Back to Code

Line 1: Forward slash. 2 asterisks. Line 2: Javadoc comment here. Line 3: Asterisk. Forward slash.

Back to Code

Line 1: Forward slash. Asterisk. Line 2: Java block comment here. Line 3: Asterisk. Forward slash.

Back to Code

Line 1: Forward slash. 2 asterisks. Line 2: Asterisk. A Javadoc comment here. Line 3: Asterisk. A second Javadoc comment here. Line 3: Asterisk. Dot, dot, dot, dot. Line 4: Asterisk. Forward slash.

Back to Code

Program code. The code contains the keywords public, class, and private. Line 1: public class sky. Line 2: left brace. Line 3, indented once: private Color color, semicolon. Line 4, indented once: public sky, left parentheses, Color c, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: color equals c, semicolon. Line 7, indented once: right brace. Line 8: right brace.

Back to Code

public Color get Color, left parentheses, right parentheses.

Back to Code

public void set Color, left parentheses, Color c, right parentheses.

Back to Code

Line 1: public static double foo 1, left parentheses, String s, right parentheses. Line 2: public String foo 2, left parentheses, char c, right parentheses.

Back to Code

Airplane dot foo 3, left parentheses, 34.6, right parentheses, semicolon.

Back to Code

Line 1: Airplane a equals new Airplane, left parentheses, right parentheses, semicolon. Line 2: int n equals a dot foo 4, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon.

Back to Code

enum Seasons, left brace, Winter, comma, Spring, comma, Summer, comma, Fall, right brace, semicolon.

Back to Code

System dot out dot print l n, left parentheses, Seasons dot Sprint dot ordinal, left parentheses, right parentheses, right parentheses, semicolon.

Back to Code

Line 1: 2 forward slashes. declare grade here. Line 2: 2 forward slashes. declare letter Grade here.

Back to Code

2 forward slashes. declare federal tax rate constant, semicolon, value is 0.07.

Back to Code

2 forward slashes. your default constructor code goes here.

Back to Code

2 forward slashes. your constructor code goes here.

Back to Code

2 forward slashes. your code goes here.

Back to Code

2 forward slashes. your code goes here.

Back to Code

2 forward slashes. your code goes here.

Back to Code

2 forward slashes. your code goes here.

Back to Code

3 forward slashes. your code goes here.

Back to Code

4 forward slashes. your code goes here.

Back to Code

Program code. The code contains the following keywords: public, class, private, String, double, boolean, static, final, void. Line 1: public class Gift. Line 2: left brace. Line 3, indented once: private String description, semicolon. Line 4, indented once: private double price, semicolon. Line 5, indented once: private String occasion, semicolon. Line 6, indented once: private boolean taxable, semicolon. Line 7: blank. Line 8, indented once: public static final double TAX underscore RATE equals 0.05, semicolon. Line 9: blank. Line 10, indented once: public Gift, left parentheses, String d, comma, double p, comma, String o, comma, boolean t, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: description equals d, semicolon. Line 13, indented twice: price equals p, semicolon. Line 14, indented twice: occasion equals o, semicolon. Line 15, indented twice: taxable equals t, semicolon. Line 16, indented once: right brace. Line 17, indented once: public void set Price, left parentheses, double p, right parentheses. Line 18, indented once: left brace. Line 19, indented twice: price equals p, semicolon. Line 20, indented once: right brace. Line 21, indented once: public void set Taxable, left parentheses, boolean t, right parentheses. Line 22, indented once: left brace. Line 23, indented twice: taxable equals t, semicolon. Line 24, indented once: right brace. Line 25: right brace.

Back to Code

Program code. The code contains the following keywords: public, void, return. Line 1: public void get Price, left parentheses, right parentheses. Line 2: left brace. Line 3, indented once: return price, semicolon. Line 4: right brace.

Back to Code

Program code. The code contains the following keywords: public, void, String. Line 1: public void set Occasion, left parentheses, String occasion, right parentheses. Line 2: left brace. Line 3, indented once: occasion, equals, occasion, semicolon. Line 4: right brace.

Back to Code

Program code. The code contains the following keywords:

public, String. Line 1: public String to String, left parentheses, right parentheses. Line 2: left brace. Line 3, indented once: System dot out dot print l n, left parentheses, double quotes, description, equals, double quotes, plus, description, right parentheses, semicolon. Line 4, indented once: System dot out dot print l n, left parentheses, double quotes, price, equals, double quotes, plus price, right parentheses, semicolon. Line 5, indented once: System dot out dot print l n, left parentheses, double quotes, occasion, equals, double quotes, plus, occasion, right parentheses, semicolon. Line 6, indented once: System dot out dot print l n, left parentheses, double quotes, taxable, equals, double quotes, plus, taxable, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, boolean, return, this. Line 1: public boolean equals, left parentheses, Object g, right parentheses. Line 2: left brace. Line 3, indented once: return, left parentheses, this, 2 equals sign, g, right parentheses, semicolon. Line 4: right brace.

Back to Code

Program code. The code contains the following keywords: pubilc, void, double. Line 1: public void set Tax Rate, left parentheses, double new Tax Rate, right parentheses. Line 2: left brace. Line 3, indented once: TAX underscore

RATE equals new Tax Rate, semicolon. Line 4: right brace.

Back to Code

Program code. The code contains the following keywords: public, double, return. Line 1: public double calc Tax, left parentheses, TAX underscore RATE, right parentheses. Line 2: left brace. Line 3, indented once: return, left parentheses, TAX underscore RATE asterisk price, right parentheses, semicolon. Line 4: right brace.

Back to Code

Line 1: Gift g equals new Gift, left parentheses, double quotes, radio, double quotes, comma, 59.99, comma, double quotes, Birthday, double quotes, comma, false, right parentheses, semicolon. Line 2: Gift dot set Price, left parentheses, 99.99, right parentheses, semicolon.

Back to Code

Line 1: Gift g equals new Gift, left parentheses, double quotes, radio, double quotes, comma, 59.99, comma, double quotes, Birthday, double quotes, comma, false, right parentheses, semicolon. Line 2: g dot set Taxable, left parentheses, right parentheses, equals, true, semicolon.

Back to Code

enum Months equals, left brace, double quotes, January, double quotes, comma, double quotes, February, double quotes, comma, double quotes, March, double quotes, right brace, semicolon.

Back to Code

Program code. The code contains the following keywords: public, class, private, char. Line 1: public class Grade. Line 2: left brace. Line 3, indented once: private char letter Grade, semicolon. Line 4, indented twice: public Grade, left parentheses, char, l g, right parentheses.

Back to Code

Program code. The code contains the following keywords: public, char, void, return. Line 5, indented once: left brace. Line 6, indented twice: letter Grade equals l g, semicolon. Line 7, indented once: right brace. Line 8, indented once: public char get Letter Grade, left parentheses, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: return letter Grade, semicolon. Line 11, indented once: right brace. Line 12, indented once: public void set Letter Grade, left parentheses, char l g, right parentheses. Line 13, indented once: left brace. Line 14, indented twice: letter Grade equals l g, semicolon. Line 15, indented once: right brace. Line 16: right brace.

Back to Code

Line 1: Grade g equals new Grade, left parentheses, single quote, uppercase b, single quote, right parentheses, semicolon. Line 2: g dot letter Grade equals, single quote, uppercase a, single quote, semicolon. 2 forward slashes, line 10.

Back to Code

Line 1: Grade Client dot java, colon, 10, colon, error, colon, letter Grade has private access. Line 2: in Grade. Line 3: g dot letter Grade equals, single quote, uppercase a, single quote, semicolon, tab spaces, 2 forward slashes, line 10. Line 4: a caret sign under the dot in Line 3. Line 4: 1 error.

Back to Code

Grade g equals new Grade, left parentheses, double quotes, uppercase a, double quotes, right parentheses, semicolon, 2 forward slashes, line 10.

Back to Code

Line 1: Grade Client dot java, colon, 10, colon, error, colon, incompatible types, colon, String. Line 2: cannot be converted to char. Line 3: Grade g equals new Grade, left parentheses, double quotes, uppercase a, double quotes, right parentheses, semicolon, 2 forward slashes, line 10. Line 4: a caret sign under the first double quotes in Line 3. Line 5: Note, colon, Some messages have been simplified, semicolon, recompile with. Line 6: hyphen X d i a g s, colon, verbose to get full output. Line 7: 1 error.

Back to Code

Program code. The code contains the following keywords: public, class, private, char. Line 1: public class Grade. Line 2: left brace. Line 3, indented once: private char letter Grade, semicolon. Line 4, indented twice: publiconce char Grade, left parentheses, char, start Letter, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: letter Grade equals start Letter, semicolon. Line 7, indented once: right brace, 2 forward slashes, line 8. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: public, class, private, char, String, return. Line 1: public class Grade. Line 2: left brace, tab space, private char letter Grade, semicolon. Line 3, indented once: public Grade, left parentheses, char l g, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: letter Grade equals l g, semicolon. Line 6, indented once: right brace. Line 7, indented once: public String to String, left parentheses, right parentheses, tab spaces, 2 forward slashes, line 10. Line 8, indented once: left brace, 2 forward slashes, line 11. Line 9, indented twice: return letter Grade, semicolon, tab space, 2 forward slashes, line 12. Line 10, indented once: right brace, 2 forward slashes, line 13. Line 11: right brace.

Back to Code

Line 1: Grade, dot, java, colon, 12, colon, error, colon, incompatible types, colon, char cannot be. Line 2: converted to String. Line 3: return letter Grade, semicolon, 2 forward slashes, line 12. Line 4: a caret sign under l of letter in Line 3. Line 5: 1 error.

Back to Code

Program code. The code contains the following keywords: public, class, private, char, String, return. Line 1: public class Grade. Line 2: left brace. Line 3, indented once: private char letter Grade, semicolon. Line 4, indented once: public Grade, left parentheses, char l g, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: letter Grade equals l g, semicolon. Line 7, indented once: right brace. Line 8, indented once: public String to String, left parentheses, right parentheses, 2 forward slashes, line 10. Line 9, indented once: left brace, 2 forward slashes, line 11. Line 10, indented twice: return l g, semicolon. 2 forward slashes, line 12. Line 11, indented once: right brace, 2 forward slashes, line 13. Line 12: right brace.

Back to Code

Line 1: Grade dot java, colon, 12, colon, error, colon, cannot find symbol. Line 2, indented once: return l g, semicolon, 2 forward slashes, line 12. Line 3: a caret sign under l of l g in Line 2. Line 4, indented twice: symbol, colon, variable l g. Line 5, indented twice: location, colon, class Grade. Line 6: 1 error.

Back to Code

Program code. The code contains the following keywords: public, class, private, int, return. Line 1: public class Grade. Line 2: left brace. Line 3, indented once: private int number Grade, semicolon. Line 4, indented once: public Grade, left parentheses, int number Grade, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: number Grade equals number Grade, semicolon. Line 7, indented once: right brace. Line 8, indented once: public int get Grade, left parentheses, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: return number Grade, semicolon. Line 11, indented once: right brace. Line 12: right brace.

Back to Code

Line 1: Grade g 1 equals new Grade, left parentheses, 95, right parentheses, semicolon. Line 2: System dot out dot print l n, left parentheses, g 1 dot get Grade, left parentheses, right parentheses, right parentheses, semicolon.

Back to Code

enum Seasons, left brace, Winter, comma, Spring, comma, Summer, comma, Fall, right brace, semicolon.

Back to Code

Line 1: Seasons s equals Seasons dot Spring, semicolon. Line 2: if, left parentheses, s dot equals, left parentheses, Winter, right parentheses, right parentheses, 2 forward slashes, line 10. Line 3, indented once: System dot out dot print l n, left parentheses, double quotes, It is cold, double quotes, right parentheses, semicolon. Line 4: else. Line 5, indented once: System dot out dot print l n, left parentheses, double quotes, The weather is fine, double quotes, right parentheses, semicolon.

Back to Code

Line 1: Test dot java, colon, 10, error, colon, cannot find symbol. Line 2, indented once: if, left parentheses, s dot equals, left parentheses, Winter, right parentheses, right parentheses, 2 forward slashes, line 10. Line 3: a caret sign under e of Winter. Line 4, indented once: symbol, colon, variable Winter. Line 5, indented once: location, colon, class Test. Line 6: 1 error.

Back to Code

enum Seasons, left brace, Winter, comma, Spring, comma, Summer, comma, Fall, right brace, semicolon.

Back to Code

Seasons dot Fall equals Autumn, semicolon, 2 forward slashes, line 10.

Back to Code

Line 1: Test dot java, colon, 10, colon, error, colon, cannot assign a value to final variable. Line 2: Fall. Line 3, indented once: seasons dot Fall, equals, Autumn, semicolon, 2 forward slashes, line 10. Line 4: a caret sign under the dot in Line 3. Line 5: Test dot java, colon, 10, colon, cannot find symbol. Line 6, indented once: Seasons dot Fall equals Autumn, semicolon, 2 forward slashes, line 10. Line 7: a caret sign under uppercase A in Line 6. Line 8, indented once: symbol, colon, variable Autumn. Line 9, indented once: location, colon, class Test. Line 10: 2 errors.

Back to Code

int number equals Utility dot get Int, left parentheses, double quotes, Enter an int, double quotes, comma, single quote, right angled bracket, single quote, right parentheses, semicolon.

Back to Code

Line 1: System dot out dot print l n, left parentheses, double quotes, Enter an int, right angled bracket, double quotes, right parentheses, semicolon. Line 2: Scanner scan equals new Scanner, left parentheses, System dot in, right parentheses, semicolon. Line 3: int number equals scan dot next Int, left parentheses, right parentheses, semicolon.

Back to Code

Line 1: int temp equals array, left square bracket, 3, right square bracket, semicolon. 2 forward slashes, line 1. Line 2: array, left square bracket, 3, right square bracket, equals, array, left square bracket, 6, right square bracket, semicolon. 2 forward slashes, line 2. Line 3: array, left square bracket, 6, right square bracket, equals, temp, semicolon. 2 forward slashes, line 3.

Back to Code

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 23. Index: 0. Column 2. Value: 45. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 33. Index: 3. Column 5. Value: 78. Index: 4. Column 6. Value: 90. Index: 5. Column 7. Value: 82. Index: 6. Column 8. Value: 80. Index: 7. Column 9. Value: 90. Index: 8. Column 10. Value: 66. Index: 9. Columns 4 and 7 are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 23. Index: 0. Column 2. Value: 45. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 33. Index: 3. Column 5. Value: 78. Index: 4. Column 6. Value: 90. Index: 5. Column 7. Value: 82. Index: 6. Column 8. Value: 80. Index: 7. Column 9. Value: 90. Index: 8. Column 10. Value: 66. Index: 9. Columns 4 and 7 are shaded. The value in variable temp is 33.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 23. Index: 0. Column 2. Value: 45. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 82. Index: 3. Column 5. Value: 78. Index: 4. Column 6. Value: 90. Index: 5. Column 7. Value: 82. Index: 6. Column 8. Value: 80. Index: 7. Column 9. Value: 90. Index: 8. Column 10. Value: 66. Index: 9. Columns 4 and 7 are shaded. The value in variable temp is 33.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 23. Index: 0. Column 2. Value: 45. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 82. Index: 3. Column 5. Value: 78. Index: 4. Column 6. Value: 90. Index: 5. Column 7. Value: 33. Index: 6. Column 8. Value: 80. Index: 7. Column 9. Value: 90. Index: 8. Column 10. Value: 66. Index: 9. Columns 4 and 7 are shaded. The value in variable temp is 33.

Back to Table

Line 1: for i equals 1 to last array index by 1. Line 2, indented once: j equals i. Line 3, indented once: temp equals element at index i. Line 4, indented once: while, left parentheses, j exclamation mark, equals, 0, and value of element at index j minus 1, right angled bracket, temp, right parentheses. Line 5, indented twice: shift element at index j minus 1 to the right. Line 6, indented twice: decrement j by 1. Line 7, indented once: assign value stored in temp to element at index j.

Back to Code

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 17. Index: 0. Column 2. Value: 26 in bold font. Index: 1. Column 3. Value: 5. Index: 2. Column 4. Value: 12. Index: 3. Column 1 is marked Inserted element. Columns 2 to 4 are shaded. Columns 2 to 4 are also marked Uninserted elements. A caption pointing to the index of Column 2 reads: j, left parentheses, inner loop counter, right parentheses. The value stored in temp is 26.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 17. Index: 0. Column 2. Value: 26. Index: 1. Column 3. Value: 5 in bold font. Index: 2. Column 4. Value: 12. Index: 3. Columns 1 and 2 are marked Inserted elements. Columns 3 and 4 are shaded. Columns 3 and 4 are also marked Uninserted elements. A caption pointing to the index of Column 3 reads: j, left parentheses, inner loop counter, right parentheses. The value stored in temp is 5.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 17. Index: 0. Column 2. Value: 26. Index: 1. Column 3. Value: 26. Index: 2. Column 4. Value: 12. Index: 3. Columns 1 and 2 are marked Inserted elements. Columns 3 and 4 are shaded. Columns 3 and 4 are also marked Uninserted elements. A caption pointing to the index of Column 2 reads: j, left parentheses, inner loop counter, right parentheses. The value stored in temp is 5.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 17. Index: 0. Column 2. Value: 17. Index: 1. Column 3. Value: 26. Index: 2. Column 4. Value: 12. Index: 3. Columns 1 and 2 are marked Inserted elements. Columns 3 and 4 are shaded. Columns 3 and 4 are also marked Uninserted elements. A caption pointing to the index of Column 1 reads: j, left parentheses, inner loop counter, right parentheses. The value stored in temp is 5.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 5. Index: 0. Column 2. Value: 17. Index: 1. Column 3. Value: 26. Index: 2. Column 4. Value: 12. Index: 3. Columns 1 to 3 are marked Inserted elements. Column 4 is shaded. Column 4 is also marked Uninserted element. A caption pointing to the index of Column 1 reads: j, left parentheses, inner loop counter, right parentheses. The value stored in temp is 5.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 5. Index: 0. Column 2. Value: 17. Index: 1. Column 3. Value: 26. Index: 2. Column 4. Value: 12 in bold font. Index: 3. Columns 1 to 3 are marked Inserted elements. Column 4 is shaded. Column 4 is also marked Uninserted element. A caption pointing to the index of Column 4 reads: j, left parentheses, inner loop counter, right parentheses. The value stored in temp is 12.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 5. Index: 0. Column 2. Value: 17. Index: 1. Column 3. Value: 26. Index: 2. Column 4. Value: 26. Index: 3. Columns 1 to 3 are marked Inserted elements. Column 4 is shaded. Column 4 is also marked Uninserted element. A caption pointing to the index of Column 3 reads: j, left parentheses, inner loop counter, right parentheses. The value stored in temp is 12.

Back to Table

The table has 2 rows labeled Value and Index. The

column entries are as follows. Column 1. Value: 5. Index: 0. Column 2. Value: 17. Index: 1. Column 3. Value: 17. Index: 2. Column 4. Value: 26. Index: 3. Columns 1 to 3 are marked Inserted elements. Column 4 is shaded. Column 4 is also marked Uninserted element. A caption pointing to the index of Column 2 reads: j, left parentheses, inner loop counter, right parentheses. The value stored in temp is 12.

Back to Table

Program code. The code contains the following keywords: public, static, void, Auto, int. Line 1: forward slash, 2 asterisks, Insertion sorts an array of Autos. Line 2: asterisk, tab spaces, at symbol, param a r r an array of Autos. Line 3: asterisk, forward slash. Line 4: public static void insertion Sort, left parentheses, Auto, left square bracket, right square bracket, a r r, right parentheses. Line 5: left brace. Line 6, indented once: Auto temp, semicolon. Line 7, indented once: int j, semicolon.

Back to Table

Program code. The code contains the following keywords: public, static, void, Auto, int. Line 1: forward slash, 2 asterisks, Insertion sorts an array of Autos. Line 2: asterisk, tab spaces, at symbol, param a r r an array of Autos. Line 3: asterisk, forward slash. Line 4: public static void insertion Sort, left parentheses, Auto, left square bracket, right square bracket, a r r, right parentheses. Line

5: left brace. Line 6, indented once: Auto temp, semicolon. Line 7, indented once: int j, semicolon.

Back to Code

Program code. The code contains the following keywords: for, int, while. Line 8, indented once: for, left parentheses, int i equals 1, semicolon, i less than a r r dot length, semicolon, i, plus, plus, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: j equals i, semicolon. Line 11, indented twice: temp equals a r r, left square bracket, i, right square bracket, semicolon. Line 12: blank. Line 13, indented twice: while, left parentheses, j, exclamation mark, equals sign, 0, 2 ampersands, left parentheses, temp dot get Model, left parentheses, right parentheses, right parentheses, dot, compare To, left parentheses. Line 14, indented multiple times: a r r, left square bracket, j minus 1, right square bracket, dot get Model, left parentheses, right parentheses, right parentheses, less than, 0, right parentheses. Line 15, indented twice: left brace. Line 16, indented thrice: a r r, left square bracket, j, right square bracket, equals, a r r, left square bracket, j minus 1, right square bracket, semicolon. Line 17, indented thrice: j, minus, minus, semicolon. Line 18, indented twice: right brace, 2 forward slashes, end while loop. Line 19: blank. Line 20, indented twice: a r r, left square bracket, j, right square bracket, equals, temp, semicolon. Line 21: blank. Line 22, indented once: right brace. 2 forward slashes, end for loop. Line 23: right brace, 2 forward slashes, end

Insertion Sort method.

Back to Code

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. All the 17 columns are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17.

Value: 95. Index: 16. Columns 1 to 8 are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. Columns 1 to 3 are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17.

Value: 95. Index: 16. Column 3 is shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. All the 17 columns are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17.

Value: 95. Index: 16. Columns 1 to 8 are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. Columns 5 to 8 are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17.

Value: 95. Index: 16. Columns 7 and 8 are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. Column 8 is shaded.

Back to Table

The table has 2 columns labeled Outer loop counter and Indexes of element or elements at the sorted position. The row entries are as follows. Row 1. Outer loop counter: 0. Indexes of element or elements at the sorted position: n minus 1. Row 2. Outer loop counter: 1. Indexes of element or elements at the sorted position: n minus 2, n minus 1. Row 3. Outer loop counter: 2. Indexes of element or elements at the sorted position: n minus 3, n minus 2, n minus 1. Row 4. Outer loop counter: dot, dot, dot. Indexes of element or elements at the sorted position: dot, dot, dot. Row 5. Outer loop counter: n minus 3.

Indexes of element or elements at the sorted position: 2, 3, 4, dot, dot, dot, n minus 2, n minus 2, n minus 1. Row 6. Outer loop counter: n minus 2. Indexes of element or elements at the sorted position: 1, 2, 3, 4, dot, dot, dot, n minus 3, n minus 2, n minus 1.

Back to Equation

Line 1: for i equals 0 to last array index minus 1 by 1. Line 2, indented once: for j equals 0 to left parentheses, last array index minus i minus 1, right parentheses, by 1. Line 3, indented twice: if, left parentheses, 2 consecutive elements are in the wrong order, right parentheses. Line 4, indented thrice: swap them.

Back to Code

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 17 in bold font. Index: 0. Column 2. Value: 26 in bold font. Index: 1. Column 3. Value: 5. Index: 2. Column 4. Value: 2. Index: 3. The columns from 1 to 4 are shaded and they are marked Unsorted elements. An arrow pointing to Index 0 reads: j, left parentheses, inner loop counter, right parentheses.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 17. Index: 0. Column 2. Value: 26 in bold font. Index: 1.

Column 3. Value: 5 in bold font. Index: 2. Column 4. Value: 2. Index: 3. The columns from 1 to 4 are shaded and they are marked Unsorted elements. An arrow pointing to Index 1 reads: j, left parentheses, inner loop counter, right parentheses.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 17. Index: 0. Column 2. Value: 5. Index: 1. Column 3. Value: 26 in bold font. Index: 2. Column 4. Value: 2 in bold font. Index: 3. The columns from 1 to 4 are shaded and they are marked Unsorted elements. An arrow pointing to Index 2 reads: j, left parentheses, inner loop counter, right parentheses.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 17 in bold font. Index: 0. Column 2. Value: 5 in bold font. Index: 1. Column 3. Value: 2 in bold font. Index: 2. Column 4. Value: 26. Index: 3. The columns from 1 to 3 are shaded and they are marked Unsorted subarray. Column 4 has the sorted element. An arrow pointing to Index 0 reads: j, left parentheses, inner loop counter, right parentheses.

Back to Table

The table has 2 rows labeled Value and Index. The

column entries are as follows. Column 1. Value: 5. Index: 0. Column 2. Value: 17 in bold font. Index: 1. Column 3. Value: 2 in bold font. Index: 2. Column 4. Value: 26. Index: 3. The columns from 1 to 3 are shaded and they are marked Unsorted subarray. Column 4 has the sorted element. An arrow pointing to Index 1 reads: j, left parentheses, inner loop counter, right parentheses.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 5 in bold font. Index: 0. Column 2. Value: 2 in bold font. Index: 1. Column 3. Value: 17. Index: 2. Column 4. Value: 26. Index: 3. The columns 1 and 2 are shaded and they are marked Unsorted subarray. Columns 3 and 4 are marked sorted elements. An arrow pointing to Index 0 reads: j, left parentheses, inner loop counter, right parentheses.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 2 in bold font. Index: 0. Column 2. Value: 5 in bold font. Index: 1. Column 3. Value: 17. Index: 2. Column 4. Value: 26. Index: 3. Column 1 is shaded. An arrow pointing to column 1 reads: Automatically sorted. Columns 2 to 4 are marked Sorted elements.

Back to Table

Line 1: read first survey. Line 2: while, left parentheses, not end of surveys, right parentheses.

Back to Code

Line 3: left brace. Line 4, indented once: if answer is 0. Line 5, indented twice: increment counter 0. Line 6, indented once: else if answer is 1. Line 7, indented twice: increment counter 1. Line 8, indented once: else if answer is 2. Line 9, indented twice: increment counter 2. Line 10, indented once: else if answer is 3. Line 11, indented twice: increment counter 3. Line 12, indented once: read next survey. Line 13: right brace.

Back to Code

roll Count, left square bracket, my Roll minus 1, right square bracket, plus, plus, semicolon.

Back to Code

The syntax is as follows. access Modifier return Type method Name, left parentheses, left parentheses, 0 to many parameters, comma, right parentheses, data Type, dot, dot, dot, last Parameter, right parentheses.

Back to Code

Line 1: double, left square bracket, right square bracket, a, equals, left brace, 12.5, comma, 48.3, comma, 65.0, right brace, semicolon. Line 2: System dot out dot print l n, left parentheses, a, left square bracket, 1, right square bracket, right parentheses, semicolon.

Back to Code

Line 1: int, left square bracket, right square bracket, a, equals, new int, left square bracket, 6, right square bracket, semicolon. Line 2: System dot out dot print l n, left parentheses, a, left square bracket, 4, right square bracket, right parentheses, semicolon.

Back to Code

Line 1: double, left square bracket, right square bracket, a, equals, left brace, 12.5, comma, 48.3, comma, 65.0, right brace, semicolon. Line 2: System dot out dot print l n, left parentheses, a dot length, right parentheses, semicolon.

Back to Code

Line 1: int, left square bracket, right square bracket, a, equals, left brace, 12, comma, 48, comma, 65, right brace, semicolon. Line 2: blank. Line 3: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 4, indented once: System dot out dot print l n, left parentheses, a, left square bracket, i, right square bracket, right parentheses, semicolon.

Back to Code

Line 1: int, left square bracket, right square bracket, a, equals, left brace, 12, comma, 48, comma, 65, right brace, semicolon. Line 2: blank. Line 3: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 4, indented once: System dot out dot print l n, left parentheses, double quotes, a, left square bracket, double quotes, i, plus, double quotes, right square bracket, equals, double quotes, plus, a, left square bracket, i, right square bracket, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: int, for. Line 1: int s equals 0, semicolon. Line 2: int, left square bracket, right square bracket, a, equals, left brace, 12, comma, 48, comma, 65, right brace, semicolon. Line 3: blank. Line 4: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 5, indented once: s plus equals a, left square bracket, i, right square bracket, semicolon. Line 6: System dot out dot print l n, left parentheses, double quotes, s equals, blank space, double quotes, plus s, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: int, new, for. Line 1: int, left square bracket, right square bracket, a, equals, new int, left square bracket, 18, right square bracket, semicolon. Line 2: blank. Line 3: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 4: a, left square bracket, i, right square bracket, equals, i plus 10, semicolon. Line 5: blank. Line 6: System dot out dot print l n, left parentheses, a, left square bracket, 4, right square bracket, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: double, for, int, if. Line 1: double, left square bracket, right square bracket, a, equals, left brace, 12.3, comma, 99.6, comma, 48.2, comma, 65.8, right brace, semicolon. Line 2: double temp equals a, left square bracket, 0, right square bracket, semicolon. Line 3: blank. Line 4: for, left parentheses, int, i equals 1, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 5: left brace. Line 6, indented once: if, left parentheses, a, left square bracket, i, right square bracket, greater than, temp, right parentheses. Line 7, indented twice: temp equals a, left parentheses, i, right parentheses, semicolon. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: int, for. Line 1: int, left square bracket, right square bracket, a, equals, left brace, 12, comma, 48, comma, 65, comma, 23, right brace, semicolon. Line 2: int temp, equals, a, left square bracket, 1, right square bracket, semicolon. Line 3: a, left square bracket, 1, right square bracket, equals, a, left square bracket, 3, right square bracket, semicolon. Line 4: a, left square bracket, 3, right square bracket, equals, temp, semicolon.

Back to Code

Program code. The code contains the following keywords: public, int, for, if, return. Line 1: public int foo, left parentheses, int, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: int temp equals 0, semicolon. Line 4, indented once: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i plus, plus, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: if, left parentheses, a, left square bracket, i, right square bracket, 2 equals sign, 5, right parentheses. Line 7, indented thrice: temp, plus, plus, semicolon. Line 8, indented once: right brace. Line 9, indented once: return temp, semicolon. Line 10: right brace.

Back to Code

Program code. The code contains the following keywords: public, int, for, if, return. Line 1: public int foo, left parentheses, int, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: if, left parentheses, a, left square bracket, i, right square bracket, 2 equals sign, 10, right parentheses. Line 6, indented thrice: return i, semicolon. Line 7, indented once: right brace. Line 8, indented once: return negative 1, semicolon. Line 9: right brace.

Back to Code

Program code. The code contains the following keywords: public, boolean, int, for, if, return, false, true. Line 1: public boolean foo, left parentheses, int, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: if, left parentheses, a, left square bracket, i, right square bracket, less than 0, right parentheses. Line 6, indented thrice: return false, semicolon. Line 7, indented once: right brace. Line 8, indented once: return true, semicolon. Line 9: right brace.

Back to Code

Program code. The code contains the following keywords: public, String, new, for, int, return. Line 1: public String, left square bracket, right square bracket, foo, left parentheses, String, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: String, left square bracket, right square bracket, temp, equals, new String, left square bracket, a dot length, right square bracket, semicolon. Line 4, indented once: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i plus, plus, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: temp, left square bracket, i, right square bracket, equals, a, left square bracket, i, right square bracket, dot, to Lower Case, left parentheses, right parentheses, semicolon. Line 7, indented once: right brace. Line 8, indented once: return temp, semicolon. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: public, boolean, String, new, for, int, if, else, true, false, return. Line 1: public boolean, left square bracket, right square bracket, foo, left parentheses, String, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: boolean, left square bracket, right square bracket, temp, equals, new boolean, left square bracket, a dot length, right square bracket, semicolon. Line 4, indented once: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i plus, plus, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: if, left parentheses, a, left square bracket, i, right square bracket, dot index Of, left parentheses, double quotes, at symbol, double quotes, right parentheses, exclamation mark, equals, negative 1, right parentheses. Line 7, indented thrice: temp, left square bracket, i, right square bracket, equals, true, semicolon. Line 8, indented twice: else. Line 9, indented thrice: temp, left square bracket, i, right square bracket, equals, false, semicolon. Line 10, indented once: right brace. Line 11, indented once: return temp, semicolon. Line 12: right brace.

Back to Code

Program code. The code contains the following keywords: int, new, for. Line 1: int, left square bracket, right square bracket, a, equals, new int, left square bracket, 25, right square bracket, semicolon. Line 2: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 3: left brace. Line 4, indented once: 2 forward slashes, your code goes here. Line 5: right brace.

Back to Code

Program code. The code contains the following keywords: double, for, int. Line 1: double, left square bracket, right square bracket, a, equals, left brace, 45.2, comma, 13.1, comma, 12.8, comma, 87.4, comma, 99.0, comma, 100.1, comma, 43.8, comma, 2.4, right brace, semicolon. Line 2: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 3: left brace. Line 4, indented once: 2 forward slashes, your code goes here. Line 5: right brace.

Back to Code

Program code. The code contains the following keywords: int, double, for. Line 1: int, left square bracket, right square bracket, a, equals, left brace, 45, comma, 13, comma, 12, comma, 87, comma, 99, comma, 100, comma, 43, comma, 2, right brace, semicolon. Line 2: blank. Line 3: double average equals 0.0, semicolon. Line 4: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 5: left brace. Line 6, indented once: 2 forward slashes, your code goes here. Line 7: right brace. Line 8: 2 forward slashes, dot, dot, dot, and your code continues here.

Back to Code

Program code. The code contains the following keywords: int, for. Line 1: int, left square bracket, right square bracket, a, equals, left brace, 3, comma, 7, comma, 9, right brace, semicolon. Line 2: int, left square bracket, right square bracket, b, equals, left brace, 2, comma, 9, comma, 4, right brace, semicolon. Line 3: int dot Product equals 0, semicolon. Line 4: blank. Line 5: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 6: left brace. Line 7, indented once: 2 forward slashes, your code goes here. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords:

int, for. Line 1: a, left square bracket, 0, right square bracket, equals, 3. Line 2: a, left square bracket, 1, right square bracket, equals, 6. Line 3: a, left square bracket, 2, right square bracket, equals, 10. Line 4: int, left square bracket, right square bracket, a, equals, left brace, 3, comma, 6, comma, 10, right brace, semicolon. Line 5: for, left parentheses, int i equals 0, semicolon, i less than a dot length, semicolon, i, plus, plus, right parentheses. Line 6: left brace. Line 7, indented once: 2 forward slashes, your code goes here. Line 8: right brace.

Back to Code

Line 1: public boolean foo, left parentheses, String, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, your code goes here. Line 4: right brace.

Back to Code

Line 1: public int foo, left parentheses, int, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, your code goes here. Line 4: right brace.

Back to Code

Line 1: public boolean foo, left parentheses, String, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, your code goes here. Line 4: right brace.

Back to Code

Line 1: public boolean, left square bracket, right square bracket, foo, left parentheses, int, left square bracket, right square bracket, a, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, your code goes here. Line 4: right brace.

Back to Code

Line 1: double, left square bracket, right square bracket, a, equals, left brace, 3.3, comma, 26.0, comma, 48.4, right brace, semicolon. Line 2: a, left square bracket, 4, right square bracket, equals, 2.5, semicolon.

Back to Code

Line 1: double, left square bracket, right square bracket, a, equals, left brace, 3.3, comma, 26.0, comma, 48.4, right brace, semicolon. Line 2: System dot out dot print l n, left parentheses, a, left square bracket, negative 1, right square bracket, right parentheses, semicolon.

Back to Code

Line 1: double, left square bracket, right square bracket, equals, left brace, 3.3, comma, 26.0, comma, 48.4, right brace, semicolon. Line 2: System dot out dot print l n, left parentheses, a, left brace, 1, right brace, right parentheses, semicolon.

Back to Code

Line 1: double, left square bracket, right square bracket, a, equals, left brace, 3.3, comma, 26.0, comma, 48.4, right brace, semicolon. Line 2: for, left parentheses, int i equals 0, semicolon, i less than or equal to a dot length, semicolon, i plus, plus, right parentheses. Line 3, indented once: System dot out dot print l n, left parentheses, a, left square bracket, i, right square bracket, right parentheses, semicolon.

Back to Code

double a, left square bracket, 3, right square bracket, equals, left brace, 3.3, comma, 26.0, comma, 48.4, right brace, semicolon.

Back to Code

Line 1: int a, left square bracket, right square bracket, equals, left brace, 3, comma, 26, comma, 48, comma, 5, right brace, semicolon. Line 2: int b, left square bracket, right square bracket, equals, left brace, 3, comma, 26, comma, 48, comma, 5, right brace, semicolon. Line 3: if, left parentheses, a, exclamation mark, equals, b, right parentheses. Line 4, indented once: System dot out dot print l n, left parentheses, double quotes, Array elements are NOT identical, double quotes, right parentheses, semicolon.

Back to Code

Line 1: int, left square bracket, right square bracket, a, equals, left brace, 3, comma, 26, comma, 48, comma, 5, right brace, semicolon. Line 2: a dot length equals 10, semicolon.

Back to Code

Line 1: int, left square bracket, right square bracket, equals, left brace, 3, comma, 26, comma, 48, comma, 5, right brace, semicolon. Line 2: System dot out dot print l n, left parentheses, double quotes, The array elements are, double quotes, plus, a, right parentheses, semicolon.

Back to Code

Line 1: Integer i 1 equals 10, semicolon. Line 2: Integer i 2 equals 15, semicolon. Line 3: Double d 1 equals 3.4, semicolon. Line 4: String s equals new String, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 5: Integer, left square bracket, right square bracket, a, equals, left brace, i 1, comma, i 2, comma, d 1, comma, s, right brace, semicolon.

Back to Code

int a, left square bracket, 6, right square bracket, equals, left brace, 2, comma, 7, comma, 8, comma, 9, comma, 11, comma, 16, right brace, semicolon, 2 forward slashes, line 26.

Back to Code

Line 1: Test dot java, colon, 26, colon, error, colon, single quote, right square bracket, single quote, expected. Line 2, indented once: int a, left square bracket, 6, right square bracket, equals sign, left brace, 2, comma, 7, comma, 8, comma, 9, comma, 11, comma, 16, right brace, semicolon, 2 forward slashes, line 26. Line 3: A caret sign under lowercase a in Line 2. Line 4: Test dot java, colon, 26, colon, error, colon, not a statement. Line 5, indented once: int a, left square bracket, 6, right square bracket, equals, left brace, 2, comma, 7, comma, 8, comma, 9, comma, 11, comma, 16, right brace, semicolon, 2 forward slashes, line 26. Line 6: a caret sign under 2 in Line 5. Line 7: Test dot java, colon, 26, colon, error, colon, single quote, semicolon, single quote, expected. Line 8, indented once: int a, left square bracket, 6, right square bracket, equals, left brace, 2, comma, 7, comma, 8, comma, 9, comma, 11, comma, 16, right brace, semicolon, 2 forward slashes, line 26. Line 9: A caret sign under the comma following 2 in Line 8. Line 10: 3 errors.

Back to Code

Program code. The code contains the keywords int and for. Line 1: int, left square bracket, right square bracket, a, equals, left brace, 2, comma, 7, comma, 8, comma, 9, comma, 11, comma, 16, right brace, semicolon, 2 forward slashes, line 26 of class Test dot java. Line 2: for, left parentheses, int i equals 0, semicolon, i less than or equal to a dot length, semicolon, i plus, plus, right parentheses, 2 forward slashes, line 27 of class Test dot java. Line 3, indented once: System dot out dot print l n, left parentheses, a, left square bracket, i, right square bracket, right parentheses, semicolon, 2 forward slashes, line 28 of class Test dot java.

Back to Code

Line 1: 2. Line 2: 7. Line 3: 8. Line 4: 9. Line 5: 11. Line 6: 16. Line 7: Exception in thread, double quotes, main, double quotes, java dot lang dot Array Index Out Of Bounds. Line 8, indented once: Exception, colon, Index 6 out of bounds for length 6. Line 9: at Test dot main, left parentheses, Test 46 dot java, colon, 28, right parentheses.

Back to Code

Program code. The code contains the following keywords: int, if, else. Line 1: int, left square bracket, right square bracket, a, equals, left brace, 1, comma, 2, comma, 3, right brace, semicolon. Line 2: int, left square bracket, right square bracket, b, equals, left brace, 1, comma, 2, comma, 3, right brace, semicolon. Line 3: if, left parentheses, a, 2 equals sign, b, right parentheses. Line 4, indented once: System dot out dot print l n, left parentheses, double quotes, Arrays are equal, double quotes, right parentheses, semicolon. Line 5: else. Line 6, indented once: System dot out dot print l n, left parentheses, double quotes, Arrays are NOT equal, double quotes, right parentheses, semicolon.

Back to Code

Line 1: int, left square bracket, right square bracket, a, equals, left brace, 1, comma, 2, comma, 3, right brace, semicolon. Line 2: System dot out dot print l n, left parentheses, a, right parentheses, semicolon.

Back to Code

The output is as follows. left square bracket, uppercase I, at symbol, lowercase f, 0, 3, 2, 6, 2, 6, 7.

Back to Code

The syntax is as follows: Line 1: access Modifier class Subclass Name extends Superclass Name. Line 2: left brace. Line 3, indented once: 2 forward slashes, class definition. Line 4: right brace.

Back to Code

public class Draw A Sprite extends Application.

Back to Code

super, left parentheses, argument list, right parentheses, semicolon.

Back to Code

call to super must be first statement in constructor.

Back to Code

this dot balance equals balance, semicolon.

Back to Code

balance has private access in Bank Account.

Back to Code

The syntax is as follows: super dot method Name, left parentheses, argument list, right parentheses.

Back to Code

The syntax is as follows: access Method abstract class Class Name.

Back to Code

The syntax is as follows: access Modifier abstract return Type method Name, left parentheses, argument list, right parentheses, semicolon.

Back to Code

The error reads: class Name is abstract, semicolon, cannot be instantiated.

Back to Code

Line 1: Forward slash, 2 asterisks, prepare To Race method. Line 2: Asterisk, at symbol param input racer type entered by the user. Line 3: Asterisk, racer types are, single quote, t, single quote, or, single quote, T, single quote, for Tortoise. Line 4: Asterisk, single quote, h, single quote, or , single quote, H, single quote, for Hare. Line 5: Asterisk, Forward slash. Line 6: private void prepare To Race, left parentheses, char input, right parentheses. Line 7: Left brace. Line 8: final int START underscore LINE equals 60, semicolon. 2 forward slashes, x position of start of race. Line 9: final int RACER underscore SPACE equals 50, semicolon. 2 forward slashes, spacing between racers. Line 10: Forward slash, 2 asterisks, 1. Asterisk, Asterisk, Asterisk, Asterisk, Asterisk, Student writes this switch statement. Line 11: Asterisk, input parameter contains the racer type. Line 12: Asterisk, entered by the user. Line 13: Asterisk, If input is, single quote, T, single quote, or, single quote, t, single quote. Line 14: Asterisk, add a Tortoise object to the Array List named racer List. Line 15: Asterisk, which is an instance variable of this class. Line 16: Asterisk, The A P I of the Tortoise constructor is, colon. Line 17: Asterisk, Tortoise, left parentheses, String I D, int x, int y , right parentheses. Line 18: Asterisk, a sample call to the constructor is. Line 19: Asterisk, new Tortoise, left parentheses, double quotes, Tortoise, double quotes, START underscore LINE, y Pos , right parentheses. Line 20: Asterisk, where START underscore LINE is a constant local variable. Line 21: Asterisk, representing the starting x position for the race. Line 22: Asterisk, and y Pos is an instance variable representing. Line 23: Asterisk, the racer, single quote, s y position. Line 24: Asterisk. Line 25: Asterisk, If input is, single quote, H, single quote, or, single quote, h, single quote. Line 26: Asterisk, add a Hare object to the Array List named racer List. Line 27: Asterisk, The A P I of the Hare constructor is, colon. Line 28: Asterisk, Hare, left parentheses, String I D, int x, int y , right parentheses. Line 29: Asterisk, a sample call to the constructor is. Line 30: Asterisk, new Hare, left parentheses, double quotes, Hare, double quotes, START underscore LINE, y P o s, right parentheses. Line 31: Asterisk, where START underscore LINE is a constant local variable. Line 32: Asterisk, representing the starting x position for the race. Line 33: Asterisk, and y Pos is an instance variable representing. Line 34: Asterisk, the racer, single quote, s y position. Line 35: Asterisk. Line 36: Asterisk, After adding a racer to the Array List racer List. Line 37: Asterisk, increment y P o s by the value of. Line 38: Asterisk, the constant local variable RACER underscore SPACE. Line 39: Asterisk. Line 40: Asterisk, if input is anything other than, single quote, T, single quote, comma, single quote, t, single quote. Line 41: Asterisk, single quote, H, single quote, or single quote, h, single quote, do nothing. Line 42: Asterisk, Forward slash. Line 43: 2 forward slashes, write your switch statement here. Line 44: Forward slash, 2 asterisks, end of student code, part 1 Asterisk, Forward slash. Line 45: Right brace. 2 forward slashes, end prepare To Race.

Back to Code

Line 1: Forward slash, 2 asterisks, get Ready method. Line 2: Asterisk, at symbol param g c Graphics Context

context. Line 3: Asterisk, draws the finish line and draws the racers. Line 4: Asterisk, Forward slash. Line 5: protected void get Ready, left parentheses, Graphics Context g c, right parentheses. Line 6: Left brace. Line 7: 2 forward slashes, draw the finish line. Line 8: finish X equals, left parentheses, int, right parentheses. canvas dot get Width, left parentheses, right parentheses hyphen 20, semicolon. Line 9: g c dot set Stroke, left parentheses, Color dot BLUE, right parentheses, semicolon. Line 10: g c dot stroke Line, left parentheses, finish X, 0, finish X, canvas dot get Height, left parentheses, right parentheses, right parentheses, semicolon. Line 11: 2 forward slashes, display racers before race begins. Line 12: Forward slash, Asterisk, 2 Asterisk, Asterisk, Asterisk, Asterisk, Asterisk, student writes this code. Line 13: Asterisk, loop through instance variable Array List racer List. Line 14: Asterisk, which contains Racer object references. Line 15: Asterisk, calling draw for each element, left parentheses, Do not call move, exclamation, right parentheses. Line 16: Asterisk, The API for draw is, colon. Line 17: Asterisk, void draw, left parentheses, Graphics Context g c, right parentheses. Line 18: Asterisk, where g c is the graphics context. Line 19: Asterisk, passed to this get Ready method. Line 20: Asterisk, Forward slash. Line 21: 2 forward slashes, student code goes here. Line 22: Forward slash, 2 asterisks, end of student code, part 2 Asterisk, Forward slash. Line 23: Right brace, 2 forward slashes, end get Ready.

Back to Code

Line 1: Forward slash, 2 asterisks, run Race method. Line 2: Asterisk, at symbol param g c Graphics Context context. Line 3: Asterisk, moves and draws racers. Line 4: Asterisk, Forward slash. Line 5: public void run Race, left parentheses, Graphics Context g c, right parentheses. Line 6: Left brace. Line 7: g c dot set Stroke, left parentheses, Color dot BLUE, right parentheses, semicolon. Line 8: g c dot stroke Line, left parentheses, finish X, 0, finish X, canvas dot get Height, left parentheses, right parentheses, right parentheses, semicolon. Line 9: Forward slash, Asterisk, 3. Asterisk, Asterisk, Asterisk, Asterisk, Asterisk, student writes this code. Line 10: Asterisk, loop through instance variable Array List racer List. Line 11: Asterisk, which contains Racer object references. Line 12: Asterisk, calling move, then draw for each element. Line 13: Asterisk, The A P I for move is, colon. Line 14: Asterisk, void move, left parentheses, right parentheses. Line 15: Asterisk, The A P I for draw is, colon. Line 16: Asterisk, void draw, left parentheses, Graphics Context g c, right parentheses. Line 17: Asterisk, where g c is the Graphics Context object. Line 18: Asterisk, passed to this run Race method. Line 19: Asterisk, Forward slash. Line 20: 2 forward slashes, student code goes here. Line 21: Forward slash, 2 asterisks, end of student code, part 3 Asterisk, Forward slash. Line 22: Right brace. 2 forward slashes, end run Race.

Back to Code

The syntax is as follows. Line 1: access Modifier interface Interface Name. Line 2: left brace. Line 3, indented twice: 2 forward slashes, body of interface. Line 4: right brace.

Back to Code

The syntax is as follows: access Modifier class Class Name extends Super Class Name implements Interface 1, comma, Interface 2, comma, dot, dot, dot.

Back to Code

The syntax is as follows: access Modifier class Class Name extends Super class Name.

Back to Code

The syntax is as follows: super, left parentheses, argument list, right parentheses, semicolon.

Back to Code

The syntax is as follows: super dot method Name, left parentheses, argument list, right parentheses.

Back to Code

The syntax is as follows. Line 1: access Modifier interface Interface Name. Line 2: left brace. Line 3, indented twice: 2 forward slashes, body of interface. Line 4: right brace.

Back to Code

The syntax is as follows: access Modifier class Class Name implements Interface Name.

Back to Code

The syntax is as follows: access Modifier class Class Name extends Super class Name implements Interface 1, comma, Interface 2, comma, dot, dot, dot.

Back to Code

Program code. The code contains the following keywords: public, void. Line 1: public void my Method, left parentheses, right parentheses. Line 2: left brace. Line 3: right brace.

Back to Code

Program code. The code contains the following keywords: public, class, private, protected, int, String, double, void, extends, char, super, return. Line 1: public class uppercase a. Line 2: left brace. Line 3, indented once: private int number, semicolon. Line 4, indented once: protected String name, semicolon. Line 5, indented once: public double price, semicolon. Line 6, indented once: public uppercase a, left parentheses, right parentheses. Line 7, indented once: left brace. Line 8, indented twice: System dot out dot print l n, left parentheses, double quotes, uppercase a, left parentheses, right parentheses, called, double quotes, right parentheses, semicolon. Line 9, indented once: right brace. Line 10, indented once: private void foo 1, left parentheses, right parentheses. Line 11, indented once: left brace. Line 12, indented twice: System dot out dot print l n, left parentheses, double quotes, A version of foo 1, left parentheses, right parentheses, called, double quotes, right parentheses, semicolon. Line 13, indented once: right brace. Line 14, indented once: protected int foo 2, left parentheses, right parentheses. Line 15, indented once: left brace. Line 16, indented twice: System dot out dot print l n, left parentheses, double quotes, A version of foo 2, left parentheses, right parentheses, called, double quotes, right parentheses, semicolon. Line 17, indented twice: return number, semicolon. Line 18, indented once: right brace. Line 19, indented once: public string foo 3, left parentheses, right parentheses. Line 20, indented once: left brace. Line 21, indented twice: System dot out dot print l n, left parentheses, double quotes, A version of foo 3, left parentheses, right parentheses, called, double quotes, right parentheses, semicolon. Line 22, indented twice: return, double quotes, Hi, double quotes, semicolon. Line 23, indented ocne: right brace. Line 24: right brace. Line 25: public class uppercase b extends uppercase a. Line 26: left brace. Line 27, indented once: private char service, semicolon. Line 28, indented once: public uppercase b, left parentheses, right parentheses. Line 29, indented once: left brace. Line 30, indented twice: super, left parentheses, right parentheses, semicolon. Line 31, indented twice: System dot out dot print l n, left parentheses, double quotes, uppercase b, left parentheses, right parentheses, called, double quotes, right parentheses, semicolon. Line 32, indented once: right brace. Line 33, indented once: public void foo 1, left parentheses, right parentheses. Line 34, indented once: left brace. Line 35, indented twice: System dot out dot print l n, left parentheses, double quotes, uppercase b version of foo 1, left parentheses, right parentheses, called, double quotes, right parentheses, semicolon. Line 36, indented once: right brace. Line 37, indented once: protected int foo 2, left parentheses, right parentheses. Line 38, indented once: left brace. Line 39, indented twice: int n equals super dot foo 2, left parentheses, right parentheses, semicolon. Line 40, indented twice: System dot out dot print l n, left parentheses, double quotes, uppercase b version of foo 2, left parentheses, right parentheses, called, double quotes, right parentheses, semicolon. Line 41, indented twice: return, left parentheses, n plus 5, right parentheses, semicolon. Line 42, indented once: right brace. Line 43, indented once: public String foo 3, left parentheses, right parentheses. Line 44, indented once: left brace. Line 45, indented twice: String temp, equals, super dot foo 3, left parentheses, right parentheses, semicolon. Line 46, indented twice: System dot out dot print l n, left parentheses, double quotes, uppercase b version of foo 3, left parentheses, right parentheses, double quotes, right parentheses, semicolon. Line 47, indented twice:

return, left parentheses, temp, plus, double quotes, blank space, foo 3, double quotes, right parentheses, semicolon. Line 48, indented once: right brace. Line 49: right brace. Line 50: public class uppercase c extends uppercase b. Line 51: left brace. Line 52, indented once: public uppercase c, left parentheses, right parentheses. Line 53, indented once: left brace. Line 54, indented twice: super, left parentheses, right parentheses, semicolon. Line 55, indented twice: System dot out dot print l n, left parentheses, double quotes, uppercase c, left parentheses, right parentheses, called, double quotes, right parentheses, semicolon. Line 56, indented once: right brace. Line 57, indented once: public void foo 1, left parentheses, right parentheses. Line 58, indented once: left brace. Line 59, indented twice: System dot out dot print l n, left parentheses, double quotes, uppercase c version of foo 1, left parentheses, right parentheses called, double quotes, right parentheses, semicolon. Line 60, indented once: right brace. Line 61: right brace.

Back to Code

uppercase b, b 1, equals, new, uppercase b, left parentheses, right parentheses, semicolon.

Back to Code

Line 1: uppercase b, b 2, equals, new uppercase b, left parentheses, right parentheses, semicolon. Line 2: b 2 dot foo 1, left parentheses, right parentheses, semicolon.

Back to Code

Line 1: uppercase b, b 3, equals, new, uppercase b, left parentheses, right parentheses, semicolon. Line 2: int n, equals, b 3, dot, foo 2, left parentheses, right parentheses, semicolon.

Back to Code

Line 1: 2 forward slashes, b 4 is a uppercase b object reference. Line 2: System dot out dot print l n, left parentheses, b 4 dot foo 3, left parentheses, right parentheses, right parentheses, semicolon.

Back to Code

uppercase c, c 1, equals, new, uppercase c, left parentheses, right parentheses, semicolon.

Back to Code

Line 1: 2 forward slashes, c 2 is a uppercase c object reference. Line 2: c 2 dot foo 1, left parentheses, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, class, private, protected, String, this, return, boolean, if, else, false, interface, int. Line 1: public class uppercase f. Line 2: left brace. Line 3, indented once: private String first, semicolon. Line 4, indented once:

protected String name, semicolon. Line 5, indented once: public uppercase f, left parentheses, right parentheses. Line 6, indented once: left brace, right brace. Line 7, indented once: public uppercase f, left parentheses, String first, comma, String name, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: this dot first, equals, first, semicolon. Line 10, indented twice: this dot name, equals, name, semicolon. Line 11, indented once: right brace. Line 12, indented once: public String get First, left parentheses, right parentheses. Line 13, indented once: left brace. Line 14, indented twice: return first, semicolon. Line 15, indented once: right brace. Line 16, indented once: public String get Name, left parentheses, right parentheses. Line 17, indented once: left brace. Line 18, indented twice: return name, semicolon. Line 19, indented once: right brace. Line 20, indented once: public String to String, left parentheses, right parentheses. Line 21, indented once: left brace. Line 22, indented twice: return, double quotes, first, colon, blank space, double quotes, plus, first, plus, double quotes, back slash t, name, double quotes, colon, blank space, double quotes, plus, name, semicolon. Line 23, indented once: right brace. Line 24, indented once: public boolean equals, left parentheses, Object f, right parentheses. Line 25, indented once: left brace. Line 26, indented twice: if, left parentheses, exclamation mark, left parentheses, f instance of uppercase f, right parentheses, right parentheses. Line 27, indented thrice: return false, semicolon. Line 28, indented twice: else. Line 29, indented twice: left brace. Line 30, indented thrice: uppercase f o b j uppercase f, equals, left parentheses, uppercase f, right parentheses, f, semicolon. Line 31, indented thrice: return, left parentheses, first dot equals, left parentheses, o b j uppercase f, dot, first, right parentheses, ampersand, ampersand, name, dot, equals, left parentheses, o b j uppercase f, dot, name, right parentheses, right parentheses, semicolon. Line 32, indented twice: right brace. Line 33, indented once: right brace. Line 34: right brace. Line 35: public interface uppercase i. Line 36: left brace. Line 37, indented once: String TYPE, equals, double quotes, human, double quotes, semicolon. Line 38, indented once: int age, left parentheses, right parentheses, semicolon. Line 39: right brace.

Back to Code

Program code. The code contains the following keywords: public, abstract, class, private, void, int, extends, return, interface, double. Line 1: public abstract class uppercase c. Line 2: left brace. Line 3, indented once: private void foo 1, left parentheses, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: System dot out dot print l n, left parentheses, double quotes, Hello foo 1, left parentheses, right parentheses, double quotes, right parentheses, semicolon. Line 6, indented once: right brace. Line 7, indented once: public abstract void foo 2, left parentheses, right parentheses, semicolon. Line 8, indented once: public abstract int foo 3, left parentheses, right parentheses, semicolon. Line 9: right brace. Line 10: public class uppercase d extends uppercase c. Line 11: left brace. Line 12, indented once: public void foo 2, left parentheses, right parentheses. Line 13, indented once: left brace. Line 14, indented twice: System dot out dot print l n, left parentheses, double quotes, Hello foo 2, left parentheses, right parentheses, double quotes, right parentheses, semicolon. Line 15, indented once: right brace. Line 16, indented once: public int foo 3, left parentheses, right parentheses. Line 17, indented once: left brace. Line 18, indented twice: return 10, semicolon. Line 19, indented once: right brace. Line 20, indented once: private void foo 4, left parentheses, right parentheses. Line 21, indented once: left brace. Line 22, indented twice: System dot out dot print l n, left parentheses, double quotes, Hello foo 4, left parentheses, right parentheses, double quotes, right parentheses, semicolon. Line 23, indented once: right brace. Line 24: right brace. Line 25: public interface uppercase i. Line 26: left brace. Line 27, indented once: double PI, equals, 3.14, semicolon. Line 28: right brace.

Back to Code

uppercase c, c 1, equals, new, uppercase c, left parentheses, right parentheses, semicolon.

Back to Code

Line 1: uppercase d, d 1, equals, new, uppercase d, left parentheses, right parentheses, semicolon. Line 2: d 1 dot foo, left parentheses, right parentheses, semicolon.

Back to Code

Line 1: uppercase c, c 2, semicolon. Line 2: c 2, equals, new, uppercase d, left parentheses, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, class, extends, void, super. Line 1: public class uppercase e, extends uppercase d. Line 2: left brace. Line 3, indented once: public void foo 4, left parentheses, right parentheses. Line 4, indented once: left brace. Line 5, indented once: super dot foo 4, left parentheses, right parentheses, semicolon. Line 6, indented once: System dot out dot print l n, left parentheses, double quotes, Hello, uppercase e, foo 4, left parentheses, right parentheses, double quotes, right parentheses, semicolon. Line 7, indented once: right brace. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: public, class, extends. Line 1: public class uppercase j extends uppercase i. Line 2: left brace. Line 3: right brace.

Back to Code

Program code. The code contains the following keywords: public, class, void. Line 1: public class uppercase k. Line 2: left brace. Line 3, indented once: public void foo, left parentheses, right parentheses, semicolon. Line 4: right brace.

Back to Code

Program code. The code contains the following keywords: public, class, extends. Line 1: public class uppercase z extends String, comma, Integer. Line 2: left brace. Line 3: right brace.

Back to Code

Line 1: uppercase z dot java, colon, 1, colon, error, colon, single quote, left brace, single quote, expected. Line 2: public class uppercase z extends String, comma, Integer. Line 3: a caret sign under the comma following String in Line 2. Line 4: 1 error.

Back to Code

Program code. The code contains the following keywords: public, abstract, class, private, int, protected, double, abstract, void. Line 1: public abstract class uppercase m. Line 2: left brace. Line 3, indented once: private int n, semicolon. Line 4, indented once: protected double p, semicolon. Line 5, indented once: public abstract void foo 1, left parentheses, right parentheses, semicolon. Line 6: right brace.

Back to Code

Program code. The code contains the following keywords: public, class, extends. Line 1: public class uppercase p, extends, uppercase m. Line 2: left brace. Line 3: right brace.

Back to Code

Line 1: uppercase p dot java, colon, 1, colon, error, colon, P is not abstract and does not override. Line 2: abstract method foo 1, left parentheses, right parentheses, in uppercase m. Line 3: public class uppercase p extends uppercase m. Line 4: a caret sign under lowercase c in Line 3. Line 5: 1 error.

Back to Code

Program code. The code contains the following keywords: public, class, extends, void. Line 1: public class uppercase p, extends, uppercase m. Line 2: left brace. Line 3, indented once: public void foo 1, left parentheses, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: System dot out dot print l n, left parentheses, double quotes, n is, colon, blank space, double quotes, plus, n, right parentheses, semicolon. Line 6, indented once: right brace. Line 7: right brace.

Back to Code

Line 1: uppercase p, dot, java, colon, 5, colon, error, colon, n has private access in uppercase m. Line 2, indented once: System dot out dot print l n, left parentheses, double quotes, n is, colon, blank space, double quotes, plus, n, right parentheses, semicolon. Line 3: a caret sign below the lowercase n ahead of the right parentheses in Line 2. Line 4: 1 error.

Back to Code

Program code. The code contains the following keywords: public, class, extends, double, void, private, int, this, super. Line 1: public class uppercase p extends uppercase m. Line 2: left brace. Line 3, indented once: public uppercase p, left parentheses, double new uppercase p, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: p equals new uppercase p, semicolon. Line 6, indented once: right brace. Line 7, indented once: public void foo 1, left parentheses, right parentheses. Line 8, indented once: left brace. Line 9, indented once: right brace. Line 10: right brace. Line 11: public class uppercase q extends uppercase p. Line 12: left brace. Line 13, indented once: private int z, semicolon. Line 14, indented once: public uppercase q, left parentheses, double new uppercase p, comma, int z, right parentheses. Line 15, indented once: left brace. Line 16, indented twice: this dot z equals z, semicolon. Line 17, indented twice: super, left parentheses, new uppercase p, right parentheses, semicolon. 2 forward slashes, line 7. Line 18, indented once: right brace. Line 19: right brace.

Back to Code

Line 1: uppercase q, dot, java, colon, 7, colon, call to super must be first statement in constructor. Line 2, indented once: super, left parentheses, new uppercase p, right parentheses, semicolon, 2 forward slashes, line 17. Line 3: a caret sign under the left parentheses. Line 4: 1 error.

Back to Code

Program code. The code contains the following keywords: public, class, private, String, return, void, this. Line 1: public class Game. Line 2: left brace. Line 3, indented once: private String description, semicolon. Line 4, indented once: public Game, left parentheses, String description, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: set Description, left parentheses, description, right parentheses, semicolon. Line 7, indented once: right brace. Line 8, indented once: public String get Description, left parentheses, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: return description, semicolon. Line 11, indented once: right brace. Line 12, indented once: public void set Description, left parentheses, String description, right parentheses. Line 13, indented once: left brace. Line 14, indented twice: this dot description equals description, semicolon. Line 15, indented once: right brace. Line 16, indented once: public String to String, left parentheses, right parentheses. Line 17, indented once: left brace. Line

18, indented twice: return, double quotes, description, colon, blank space, double quotes, plus, description, semicolon. Line 19, indented once: right brace. Line 20: right brace.

Back to Code

Program code. The code contains the following keywords: public, class, final, double, private, String, void, return, this. Line 1: public class Store. Line 2: left brace. Line 3, indented once: public final double SALES underscore TAX underscore RATE, equals, 0.06, semicolon. Line 4, indented once: private String name, semicolon. Line 5, indented once: public Store, left parentheses, String name, right parentheses. Line 6, indented once: left brace. Line 7, indented twice: set Name, left parentheses, name, right parentheses, semicolon. Line 8, indented once: right brace. Line 9, indented once: public String get Name, left parentheses, right parentheses. Line 10, indented once: left brace. Line 11, indented twice: return name, semicolon. Line 12, indented once: right brace. Line 13, indented once: public void set Name, left parentheses, String name, right parentheses. Line 14, indented once: left brace. Line 15, indented twice: this dot name equals name, semicolon. Line 16, indented once: right brace. Line 17, indented once: public String to String, left parentheses, right parentheses. Line 18, indented once: left brace. Line 19, indented twice: return, left parentheses, double quotes, name, colon, blank space, double quotes, plus, name, right parentheses, semicolon.

Line 20, indented once: right brace. Line 21: right brace.

Back to Code

The magnified text reads: Look It Up.

Back to Figure

The syntax is as follows. Line 1: try. Line 2: left brace. Line 3, indented once: 2 forward slashes, code that might generate an exception. Line 4: right brace. Line 5: catch, left parentheses, Exception Class exception O b j R e f, right parentheses. Line 6: left brace. Line 7, indented once: 2 forward slashes, code to recover from the exception. Line 8: right brace.

Back to Code

The syntax is as follows. Line 1: try. Line 2: left brace. Line 3, indented once: 2 forward slashes, code that might generate an exception. Line 4: right brace. Line 5: catch, left parentheses, Exception 1 Class e 1, right parentheses. Line 6: left brace. Line 7, indented once: 2 forward slashes, code to handle an Exception Class exception. Line 8: right brace. Line 9: dot, dot, dot. Line 10: catch, left parentheses, Exception N Class e N, right parentheses. Line 11: left brace. Line 12, indented once: 2 forward slashes, code to handle an Exception N Class exception. Line 13: right brace. Line 14: finally. Line 15: left brace. Line 16, indented once: 2 forward slashes, code to execute regardless of whether an exception occurs. Line 17: right brace.

Back to Code

Line 1: File f, equals, new File, left parentheses, double quotes, movies dot t x t, double quotes, right parentheses, semicolon. Line 2: Scanner file, equals, new Scanner, left parentheses, f, right parentheses, semicolon.

Back to Code

Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, movies dot t x t, double quotes, right parentheses, right parentheses, semicolon.

Back to Code

Line 1: try, left parentheses, Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, movies dot t x t, double quotes, right parentheses, right parentheses, right parentheses. Line 2: left brace. Line 3: dot, dot, dot. Line 4: right brace. Line 5: catch, left parentheses, dot, dot, dot, right parentheses. Line 6: left brace. Line 7: dot, dot, dot. Line 8: right brace.

Back to Code

Line 1: Scanner file, semicolon. Line 2: try. Line 3: left brace. Line 4, indented once: file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, movies dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 5, indented once: dot, dot, dot. Line 6: right brace. Line 7: catch, left parentheses, dot, dot, right parentheses. Line 8: left brace. Line 9, indented once: dot, dot. Line 10: right brace. Line 11: finally. Line 12: left brace. Line 13, indented once: file dot close, left parentheses, right parentheses, semicolon. Line 14: right brace.

Back to Code

Line 1: A A 123, comma, B W I, comma, S F O, comma, 235, comma, 239.5. Line 2: A A 200, comma, B O S, comma, J F K, comma, 150, comma, 89.3. Line 3: A A 900, comma, L A X, comma, C H I, comma, 201, comma, 201.8. Line 4: Dot, dot, dot.

Back to Code

Line 1: while, left parentheses, parse dot has Next, left parentheses, right parentheses, right parentheses. Line 2, indented once: System dot out dot print l n, left parentheses, parse dot next, left parentheses, right parentheses, right parentheses, semicolon.

Back to Code

Line 1: A A 123, comma, B W I, comma, S F O, comma, 235, comma, 239.5. Line 2: A A 200, comma, B O S, comma, J F K, comma, 150, comma, 89.3. Line 3: A A 900, comma, L A X, comma, C H I, comma, 201, comma, 201.8. Line 4: Dot, dot, dot.

Back to Code

The syntax is as follows: record Record Class Name, left parentheses, parameters List, right parentheses, left brace, right brace.

Back to Code

The A P I is as follows: public void animate, left parentheses, String current Transaction, comma, double current Amount, comma, double current Balance, right parentheses.

Back to Code

animate, left parentheses, transaction Name, comma, amount, comma, balance, right parentheses, semicolon.

Back to Code

city Stream, equals, city Stream dot filter, left parentheses, city Predicate, right parentheses, semicolon.

Back to Code

Line 1: Start With B Predicate, left angled bracket, String, right angled bracket, city Predicate. Line 2, indented once: equals sign, new Start With B Predicate, left angled bracket, String, right angled bracket, left parentheses, right parentheses, semicolon. Line 3: city Stream, equals, city Stream dot filter, left parentheses, city Predicate, right parentheses, semicolon.

Back to Code

city Stream, equals, city Stream dot filter, left parentheses, city, hyphen, right angled bracket, city dot char At, left parentheses, 0, right parentheses, equals, equals, single quote, uppercase b, single quote, right parentheses, semicolon.

Back to Code

city dot char At, left parentheses, 0, right parentheses, equals, equals, single quote, uppercase b, single quote, right parentheses.

Back to Code

long count, equals, city Stream dot count, left parentheses, right parentheses, semicolon.

Back to Code

long count, equals, city Stream, dot, filter, left parentheses, city, hyphen, right angled bracket, city dot char At, left parentheses, 0, right parentheses, equals, equals, single quote, uppercase b, single quote, right parentheses, dot, count, left parentheses, right parentheses, semicolon.

Back to Code

data, hyphen, right angled bracket, Double dot parse Double, left parentheses, data, right parentheses.

Back to Code

cost, hyphen, right angled bracket, cost, right angled bracket, 0.0.

Back to Code

h t t p s, colon, forward slash, forward slash, raw dot git hub user content dot com, forward slash, p rust, forward slash, wikipedia, hyphen, movie, hyphen, data, forward slash, master, forward slash, movies dot j s o n.

Back to Code

Program code. The code contains the following keywords: String, try, new, catch. Line 1: String GIT HUB, equals, double quotes, h t t p s, colon, 2 forward slashes, raw dot git hub user content dot com, forward slash, double quotes, semicolon. Line 2: String resource equals GIT HUB plus, double quotes, p rust, forward salsh, wikipedia hyphen movie hyphen data, forward slash, master, forward slash, movies dot j s o n, double quotes, semicolon. Line 3: try. Line 4: left brace. Line 5, indented once: U R L, u r l, equals, new, U R L, left parentheses, resource, right parentheses, semicolon. Line 6, indented once: Input Stream i s, equals, u r l dot open Stream, left parentheses, right parentheses, semicolon. Line 7, indented once: Scanner data Source, equals, new Scanner, left parentheses, i s, right parentheses, semicolon. Line 8: dot, dot, dot. Line 9: right brace. Line 10: catch, left parentheses, Malformed U R L Exception m u r l e, right parentheses. Line 11: left brace. Line 12: dot, dot, dot. Line 13: right brace. Line 14: catch, left parentheses, I O Exception i o e, right parentheses. Line 15: left brace. Line 16: dot, dot, dot. Line 17: right brace.

Back to Code

Line 1: left brace, double quotes, e mail, double quotes, colon, double quotes, jane 45 at gmail dot com, double quotes, comma, double quotes, age, double quotes, colon, 21, right brace. Line 2: left brace, double quotes, states, double quotes, colon, left brace, double quotes, M D, double quotes, colon, double quotes, Maryland, double quotes, comma, double quotes, N Y, double quotes, colon, double quotes, New York, double quotes, comma, double quotes, C A, double quotes, colon, double quotes, California, double quotes, right brace, right brace. Line 3: left square bracket, double quotes, New York, double quotes, comma, double quotes, London, double quotes, comma, double quotes, Buenos Aires, double quotes, right square bracket. Line 4: left brace, double quotes, countries, double quotes, colon, left square bracket, double quotes, U S A, double quotes, comma, double quotes, China, double quotes, comma, double quotes, Brazil, double quotes, right square bracket, right brace.

Back to Code

Line 1: Left brace, double quotes, title, double quotes, colon, double quotes, Destroyer, double quotes, comma, double quotes, year, double quotes, colon, 2018, comma, double quotes, cast, double quotes, colon, left square bracket, double quotes, Nicole Kidman, double quotes, comma. Line 2: double quotes, Tatiana Maslany, double quotes, comma, double quotes, Sebastian Stan, double quotes, comma, double quotes, Toby Kebbell, double quotes, comma, double quotes, Scoot. Line 3: McNairy, double quotes, right square bracket, comma, double quotes, genres, double quotes, colon, left square bracket, double quotes, Crime, double quotes, comma, double quotes, Thriller, double quotes, right square bracket, right brace.

Back to Code

Line 1: 2 forward slashes, j son is a String that contains the data in Figure 11.20. Line 2: J S O N Array j s o n Array, equals, new, J S O N Array, left parentheses, j s o n, right parentheses, semicolon. Line 3: 2 forward slashes, get the J S O N Object at index i. Line 4: J S O N Object j s o n Movie, equals, j s o n Array, dot, get J S O N Object, left parentheses, i, right parentheses, semicolon. Line 5: 2 forward slashes, retrieve the title of the current movie. Line 6: String title, equals, j s o n Movie dot get String, left parentheses, double quotes, title, double quotes, right parentheses, semicolon. Line 7: 2 forward slashes, retrieve the year of the current movie. Line 8: int year, equals, j s o n Movie dot get Int, left parentheses, double quotes, year, double quotes, right parentheses, semicolon.

Back to Code

The syntax is as follows: access Modifier transient data Type instance Variable Name.

Back to Code

The A P I is as follows: public double get Amount, left parentheses, right parentheses.

Back to Code

The A P I is as follows: public void animate, left parentheses, Transaction current Transaction, comma, double current Balance, right parentheses.

Back to Code

The method call is as follows: animate, left parentheses, transaction Object, comma, balance, right parentheses, semicolon.

Back to Code

Line 1: Withdrawal w 2, equals, new Withdrawal, left parentheses, negative 200.00, right parentheses, semicolon. Line 2: transaction List dot add, left parentheses, w 2, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, class, extends, String. Line 1: public class

Exception Name extends Existing Exception Class Name. Line 2: left brace. Line 3, indented once: public Exception Name, left parentheses, String message, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: super, left parentheses, message, right parentheses, semicolon. Line 6, indented once: right brace. Line 7: right brace.

Back to Code

Line 1: access Modifier data Type method Name, left parentheses, parameter list, right parentheses. Line 2, indented multiple times: throws Exception Name. Line 3: left brace. Line 4, indented once: if, left parentheses, parameter list is legal, right parentheses. Line 5, indented twice: 2 forward slashes, process the parameter list. Line 6, indented once: else. Line 7, indented twice: throw new Exception Name, left parentheses, double quotes, Some message here, double quotes, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: try, new, String, while, catch. Line 1: try. Line 2: left brace. Line 3, indented once: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, words dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 4, indented once: String result, equals, double quotes, double quotes, semicolon. Line 5, indented once: while, left parentheses, file dot has Next, left parentheses, right parentheses, right parentheses. Line 6, indented once: left brace. Line 7, indented twice: String s, equals, file dot next, left parentheses, right parentheses, semicolon. Line 8, indented twice: result, plus, equals, s, semicolon. Line 9, indented twice: result, plus, equals, double quotes, AND, double quotes, semicolon. Line 10, indented once: right brace. Line 11, indented once: System dot out dot print l n, left parentheses, double quotes, result is, double quotes, plus, result, right parentheses, semicolon. Line 12, indented once: file dot close, left parentheses, right parentheses, semicolon. Line 13: right brace.

Back to Code

Line 1: Scanner parse, equals, new Scanner, left parentheses, double quotes, uppercase a, blank space, uppercase b, blank space, uppercase c, blank space, uppercase d, double quotes, right parentheses, semicolon. Line 2: while, left parentheses, parse dot has Next, left parentheses, right parentheses, right parentheses. Line 3, indented once: System dot out dot print, left parentheses, parse dot next, left parentheses, right parentheses, right parentheses, semicolon.

Back to Code

Line 1: Scanner parse, equals, new Scanner, left parentheses, double quotes, uppercase a, uppercase a, colon, uppercase b, uppercase b, colon, uppercase c, uppercase c, double quotes, right parentheses, semicolon. Line 2: parse dot use Delimiter, left parentheses, double quotes, colon, double quotes, right parentheses, semicolon. Line 3: while, left parentheses, parse dot has Next, left parentheses, right parentheses, right parentheses. Line 4, indented once: System dot out dot print l n, left parentheses, parse dot next, left parentheses, right parentheses, right parentheses, semicolon.

Back to Code

Line 1: Scanner parse, equals, new Scanner, left parentheses, double quotes, one AND two AND three AND four, double quotes, right parentheses, semicolon. Line 2: parse dot use Delimiter, left parentheses, double quotes, AND, double quotes, right parentheses, semicolon. Line 3: while, left parentheses, parse dot has Next, left parentheses, right parentheses, right parentheses. Line 4, indented once: System dot out dot print l n, left parentheses, parse dot next, left parentheses, right parentheses, right parentheses, semicolon.

Back to Code

Line 1: uppercase a. Line 2: uppercase b. Line 3: uppercase c. Line 4: uppercase a. Line 5: uppercase b. Line 6: uppercase a.

Back to Code

Program code. The code contains the following keywords: try, long, catch. Line 1: try. Line 2: left brace. Line 3, indented once: Stream, left angled bracket, String, right angled bracket, data Stream, equals, Files dot lines, left parentheses, Paths dot get, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 4, indented once: long count, equals, data Stream dot filter, left parentheses, letter, hyphen, right angled bracket, letter dot char At, left parentheses, equals, equals, single quote, uppercase a, single quote, right parentheses. Line 5, indented multiple times: dot count, left parentheses, right parentheses, semicolon. Line 6, indented once: System dot out dot print l n, left parentheses, double quotes, count, equals, double quotes, plus, count, right parentheses, semicolon. Line 7: right brace. Line 8: blank. Line 9: catch, left parentheses, Invalid Path Exception i p e, right parentheses. Line 10: left brace. Line 11, indented once: System dot out dot print l n, left parentheses, i p e dot get Message, left parentheses, right parentheses, right parentheses, semicolon. Line 12: right brace. Line 13: blank. Line 14: catch, left parentheses, I O Exception i o e, right parentheses. Line 15: left brace. Line 16, indented once: System dot out dot print l n, left parentheses, i o e dot get Message, left parentheses, right parentheses, right parentheses, semicolon. Line 17: right brace. Line 18: blank. Line 19: catch, left parentheses, Security Exception s e, right parentheses. Line 20: left brace. Line 21, indented once: System dot out dot print l n, left parentheses, s e dot get Message, left parentheses, right parentheses, right parentheses, semicolon. Line 22: right brace.

Back to Code

Program code. The code contains the following keywords: try, int, new, while, String, if, catch. Line 1: try. Line 2: Left brace. Line 3, indented once: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 4: blank. Line 5, indented once: int n equals 0, semicolon. Line 6, indented once: while, left parentheses, file dot has Next, left parentheses, right parentheses, right parentheses. Line 7, indented once: left brace. Line 8, indented twice: String s, equals, file dot next Line, left parentheses, right parentheses, semicolon. Line 9: blank. Line 10, indented twice: if, left parentheses, s dot equals, left parentheses, double quotes, uppercase a, double quotes, right parentheses, right parentheses. Line 11, indented thrice: n plus, plus, semicolon. Line 12, indented once: right brace. Line 13: blank. Line 14, indented once: System dot out dot print l n, left parentheses, double quotes, The value of n is, double quotes, plus, n, right parentheses, semicolon. Line 15, indented once: file dot close, left parentheses, right parentheses, semicolon. Line 16: right brace. Line 17: catch, left parentheses, I O Exception i o e, right parentheses. Line 18: left brace. Line 19, indented once: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 20: right brace.

Back to Code

Program code. The code contains the following keywords: try, new, while, String, if, else, catch. Line 1: try. Line 2: left brace. Line 3, indented once: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 4: blank. Line 5, indented once: while, left parentheses, file dot has Next, left parentheses, right parentheses, right parentheses. Line 6, indented once: left brace. Line 7, indented twice: String s, equals, file dot next Line, left parentheses, right parentheses, semicolon. Line 8, indented twice: if, left parentheses, s dot equals, left parentheses, double quotes, uppercase a, double quotes, right parentheses, right parentheses. Line 9, indented thrice: System dot out dot print l n, left parentheses, double quotes, Excellent, double quotes, right parentheses, semicolon. Line 10, indented twice: else if, left parentheses, s dot equals, left parentheses, double quotes, uppercase b, double quotes, right parentheses, right parentheses. Line 11, indented thrice: System dot out dot print l n, left parentheses, double quotes, Good, double quotes, right parentheses, semicolon. Line 12, indented twice: else. Line 13, indented thrice: System dot out dot print l n, left parentheses, double quotes, Try to do better, double quotes, right parentheses, semicolon. Line 14, indented once: right brace. Line 15, indented once: file dot close, left parentheses, right parentheses, semicolon. Line 16: right brace. Line 17: catch, left parentheses, I O Exception i o e, right parentheses. Line 18: left brace. Line 19, indented once: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 20: right brace.

Back to Code

Program code. The code contains the following keywords: try, new, String, while, if, catch. Line 1: try. Line 2: left brace. Line 3, indented once: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 4: blank. Line 5, indented once: String s, equals, double quotes, double quotes, semicolon. Line 6, indented once: while, left parentheses, file dot has Next, left parentheses, right parentheses, right parentheses. Line 7, indented once: left brace. Line 8, indented twice: s equals file dot next Line, left parentheses, right parentheses, semicolon. Line 9, indented once: right brace. Line 10: blank. Line 11, indented once: if, left parentheses, s dot equals, left parentheses, double quotes, uppercase a, double quotes, right parentheses, right parentheses. Line 12, indented twice: System dot out dot print l n, left parentheses, double quotes, Nice finish, double quotes, right parentheses, semicolon. Line 13, indented once: file dot close, left parentheses, right parentheses, semicolon. Line 14: right brace. Line 15: catch, left parentheses, I O Exception i o e, right parentheses. Line 16: left brace. Line 17, indented once: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 18: right brace.

Back to Code

Program code. The code contains the following keywords: try, new, catch. Line 1: try. Line 2: left brace. Line 3, indented once: File Writer f w, equals, new File Writer, left parentheses, double quotes, data dot t x t, double quotes, comma, true, right parentheses, semicolon. Line 4, indented once: Print Writer p w, equals, new Print Writer, left parentheses, f w, right parentheses, semicolon. Line 5: blank. Line 6, indented once: p w dot print l n, left parentheses, double quotes, Java Illuminated, double quotes, right parentheses, semicolon. Line 7, indented once: p w dot close, left parentheses, right parentheses, semicolon. Line 8: right brace. Line 9: catch, left parentheses, I O Exception i o e, right parentheses. Line 10: left brace. Line 11, indented once: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 12: right brace.

Back to Code

Program code. The code contains the following keywords: try, new, String, for, int, if, catch. Line 1: try. Line 2: left brace. Line 3, indented once: File Writer f w, equals, new, File Writer, left parentheses, double quotes, data dot t x t, double quotes, comma, false, right parentheses, semicolon. Line 4, indented once: Print Writer p w, equals, new Print Writer, left parentheses, f w, right parentheses, semicolon. Line 5: blank. Line 6, indented once: String s, equals, double quotes, uppercase a, uppercase b, uppercase c, uppercase d, uppercase e, uppercase f, uppercase g, uppercase h, double quotes, semicolon. Line 7: blank. Line 8, indented once: for, left parentheses, int i equals 0, semicolon, i less than s dot length, left parentheses, right parentheses, semicolon, i plus, plus, right parentheses. Line 9, indented once: left brace. Line 10, indented twice: if, left parentheses, i percentage sign 2, equals, equals, 0, right parentheses. Line 11, indented thrice: p w dot print, left parentheses, s dot char At, left parentheses, i, right parentheses, right parentheses, semicolon. Line 12, indented once: right brace. Line 13, indented once: p w dot print l n, left parentheses, right parentheses, semicolon. Line 14, indented once: p w dot close, left parentheses, right parentheses, semicolon. Line 15: right brace. Line 16: catch, left parentheses, I O Exception i o e, right parentheses. Line 17: left brace. Line 18, indented once: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 19: right brace.

Back to Code

Program code. The code contains the following keywords:

try, new, true, for, int, catch. Line 1: try. Line 2: left brace. Line 3, indented once: File Writer f w, equals, new File Writer, left parentheses, double quotes, data dot t x t, double quotes, comma, true, right parentheses, semicolon. Line 4, indented once: Print Writer p w, equals, new Print Writer, left parentheses, f w, right parentheses, semicolon. Line 5: blank. Line 6, indented once: for, left parentheses, int i equals 0, semicolon, i less than 5, semicolon, i plus, plus, right parentheses. Line 7, indented twice: p w dot print l n, left parentheses, i, right parentheses, semicolon. Line 8, indented once: p w dot close, left parentheses, right parentheses, semicolon. Line 9: right brace. Line 10: catch, left parentheses, I O Exception i o e, right parentheses. Line 11: left brace. Line 12, indented once: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 13: right brace.

Back to Code

Program code. The code contains the following keywords: try, new, false, int, for, catch. Line 1: try. Line 2: left brace. Line 3, indented once: File Writer f w, equals, new File Writer, left parentheses, double quotes, data dot t x t, double quotes, comma, false, right parentheses, semicolon. Line 4, indented once: Print Writer p w, equals, new Print Writer, left parentheses, f w, right parentheses, semicolon. Line 5: blank. Line 6, indented once: int s equals 0, semicolon. Line 7, indented once: for, left parentheses, int i equals 0, semicolon, i less than

5, semicolon, i plus, plus, right parentheses. Line 8, indented once: left brace. Line 9, indented twice: s plus, equals, i, semicolon. Line 11, indented once: right brace. Line 12, indented once: p w dot print, left parentheses, double quotes, The result is, double quotes, right parentheses, semicolon. Line 13, indented once: p w dot print, left parentheses, s, right parentheses, semicolon. Line 14, indented once: p w dot close, left parentheses, right parentheses, semicolon. Line 15, indented once: right brace. Line 16: catch, left parentheses, I O Exception i o e, right parentheses. Line 17, indented once: left brace. Line 18, indented twice: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 19: right brace.

Back to Code

Program code. The code contains the following keywords: String, int, try, new, while, catch. Line 1: String s, equals, double quotes, double quotes, semicolon. Line 2: int n equals 0, semicolon. Line 3: try. Line 4: left brace. Line 5, indented once: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 6, indented once: while, left parentheses, file dot has Next, left parentheses, right parentheses, right parentheses. Line 7, indented once: left brace. Line 8, indented twice: s equals file dot next Line, left parentheses, right parentheses, semicolon. Line 9, indented twice: 2 forward slashes, your code goes here. Line 10, indented once: right brace. Line 11: right brace. Line 12: catch, left parentheses, I O Exception i o e, right parentheses. Line 13: left brace. Line 14, indented once: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 15: right brace.

Back to Code

Program code. The code contains the following keywords: new, String. Line 1: Scanner parse, equals, new Scanner, left parentheses, double quotes, uppercase a, hyphen, uppercase b, hyphen, uppercase c, hyphen, uppercase d, double quotes, right parentheses, semicolon. Line 2: parse dot use Delimiter, left parentheses, double quotes, hyphen, double quotes, right parentheses, semicolon. Line 3: String s, equals, double quotes, double quotes, semicolon. Line 4: 2 forward slashes, your code goes here. Line 5: blank. Line 6: System dot out dot print l n, left parentheses, s, right parentheses, semicolon.

Back to Code

Line 1: Java. Line 2: Illuminated, colon. Line 3: Programming. Line 4: Is Not A. Line 5: Spectator. Line 6: Sport.

Back to Code

Program code. The code contains the following keywords: try, new. Line 1: try. Line 2: left brace. Line 3, indented once: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 4, indented once: 2 forward slashes, your code goes here.

Back to Code

Program code. The code contains the following keywords: try, catch. Line 1: try. Line 2: left brace. Line 3, indented once: Stream, left angled bracket, String, right angled bracket, data Stream, equals, Files, dot, lines, left parentheses, Paths dot get, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 4, indented once: 2 forward slashes, your code goes here. Line 5: blank. Line 6: right brace. Line 6: catch, left parentheses, Invalid Path Exception i p e, right parentheses. Line 7: left brace. Line 8, indented once: System, dot, out, dot, print l n, left parentheses, i p e dot get Message, left parentheses, right parentheses, right parentheses, semicolon. Line 9: right brace. Line 10: catch, left parentheses, I O Exception i o e, right parentheses. Line 11: left brace. Line 12, indented once: System dot out dot print l n, left parentheses, i o e dot get Message, left parentheses, right parentheses, right parentheses, semicolon. Line 13: right brace. Line 14: catch, left parentheses, Security Exception s e, right parentheses. Line 15: left brace. Line 16, indented once: System dot out dot print l n, left parentheses, s e dot get Message, left parentheses, right parentheses, right parentheses, semicolon. Line 17: right brace.

Back to Code

Program code. The code contains the following keywords: try, new, String, catch. Line 1: Java Illuminated, colon, Programming Is Not A Spectator Sport. Line 2: try. Line 3: left brace. Line 4, indented once: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 5: blank. Line 6, indented once: String result, equals, double quotes, double quotes, semicolon. Line 7, indented once: 2 forward slashes, your code goes here. Line 8: blank. Line 9: right brace. Line 10: catch, left parentheses, I O Exception i o e, right parentheses. Line 11: left brace. Line 12, indented once: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 13: right brace.

Back to Code

Program code. The code contains the following keywords: try, new, String, catch. Line 1: Spectator. Line 2: Sport. Line 3: try. Line 4: left brace. Line 5, indented once: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 6: blank. Line 7, indented once: String result, equals, double quotes, double quotes, semicolon. Line 8, indented once: 2 forward slashes, your code goes here. Line 9: blank. Line 10: right brace. Line 11: catch, left parentheses, I O Exception i o e, right parentheses. Line 12: left brace. Line 13, indented once: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 14: right brace.

Back to Code

Program code. The code contains the following keywords: int, try, new, false. Line 1: int, left square bracket, right square bracket, grades, equals, left brace, 98, comma, 76, comma, 82, comma, 90, comma, 100, comma, 75, right brace, semicolon. Line 2: try. Line 3: left brace. Line 4, indented once: File Writer f w, equals, new File Writer, left parentheses, double quotes, data dot t x t, double quotes, comma, false, right parentheses, semicolon. Line 5, indented once: 2 forward slashes, your code goes here. Line 6: blank. Line 7: right brace. Line 8: 2 forward slashes, and your code continues here.

Back to Code

Program code. The code contains the following keywords: int, double, for, try, new, false, catch. Line 1: int, left square bracket, right square bracket, grades, equals, left brace, 98, comma, 76, comma, 82, comma, 90, comma, 100, comma, 75, right brace, semicolon. Line 2: double average, equals, 0.0, semicolon. Line 3: for, left parentheses, int i equals 0, semicolon, i less than grades dot length, semicolon, i plus, plus, right parentheses. Line 4: left brace. Line 5, indented once: 2 forward slashes, some of your code goes here. Line 6: right brace. Line 7: 2 forward slashes, and more code goes here. Line 8: try. Line 9: left brace. Line 10, indented once: File Writer f w, equals, new File Writer, left parentheses, double quotes, data dot t x t, double quotes, comma, false, right parentheses, semicolon. Line 11, indented once: Print Writer p w, equals, new Print Writer, left parentheses, f w, right parentheses, semicolon. Line 12, indented once: 2 forward slashes, and more code goes here. Line 13: right brace. Line 14: catch, left parentheses, I O Exception i o e, right parentheses. Line 15: left brace. Line 16, indented once: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 17: right brace.

Back to Code

Program code. The code contains the following keywords: int, double, try, new, false, catch. Line 1: int i equals 45, semicolon. Line 2: double d equals 6.7, semicolon. Line 3: try. Line 4: left brace. Line 5, indented once: File Writer, f w, equals, new File Writer, left parentheses, double quotes, data dot t x t, double quotes, comma, false, right parentheses, semicolon. Line 6, indented once: Print Writer p w, equals, new Print Writer, left parentheses, f w, right parentheses, semicolon. Line 7, indented once: 2 forward slashes, your code goes here. Line 8: right brace. Line 9: catch, left parentheses, I O Exception i o e, right parentheses. Line 10: left brace. Line 11, indented once: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 12: right brace.

Back to Code

Program code. The code contains the following keywords: float, try. Line 1: float f, equals, 13.5 f, semicolon. Line 2: try. Line 3: left brace. Line 4, indented once: 2 forward slashes, your code goes here.

Back to Code

Program code. The code contains the following keywords: new, int. Line 1: Scanner parse, equals, new Scanner, left parentheses, double quotes, 1, blank space, 2, blank space, 3, double quotes, right parentheses, semicolon. Line 2: int i, equals, parse dot next, left parentheses, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: try, new, String, catch. Line 1: try. Line 2: left brace. Line 3, indented once: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 4, indented once: String s, equals, file dot next Line, left parentheses, right parentheses, semicolon. Line 5: right brace. Line 6: catch, left parentheses, Arithmetic Exception a e, right parentheses. Line 7: left brace. Line 8, indented once:

System dot out dot print l n, left parentheses, a e dot get Message, left parentheses, right parentheses, right parentheses, semicolon. Line 9: right brace.

Back to Code

Program code. The code contains the following keywords: try, new, catch. Line 1: try. Line 2: left brace. Line 3, indented once: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 4, indented once: file dot write, left parentheses, double quotes, Hello, double quotes, right parentheses, semicolon. Line 5: right brace. Line 6: catch, left parentheses, I O Exception i o e, right parentheses. Line 7: left brace. Line 8, indented once: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 9: right brace.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, main, String, try, new. Line 1: import java dot i o dot I O Exception, semicolon. Line 2: import java dot i o dot File, semicolon. Line 3: import java dot util dot Scanner, semicolon. Line 4: public class Test. Line 5: left brace. Line 6, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 7, indented once: left brace. Line 8, indented twice:

try. 2 forward slashes, line 9. Line 9, indented twice: left brace. Line 10, indented thrice: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 11: blank. Line 12, indented thrice: String string Read, equals, file dot next Line, left parentheses, right parentheses, semicolon. Line 13, indented thrice: System dot out dot print l n, left parentheses, string Read, right parentheses, semicolon. Line 14, indented twice: right brace. Line 15, indented once: right brace. Line 16: right brace.

Back to Code

Line 1: Test dot java, colon, 9, colon, error, colon, single quote, try, single quote, without, single quote, catch, single quote, comma, single quote, finally, single quote, or resource. Line 2: declarations. Line 3, indented once: try. 2 forward slashes, line 9. Line 4: a caret sign under t in Line 3. Line 5: 1 error.

Back to Code

Program code. The code contains the following keywords: import, public, class, public, static, void, main, String, try, new. Line 1: import java dot i o dot I O Exception, semicolon. Line 2: import java dot i o dot File, semicolon. Line 3: import java dot util dot Scanner, semicolon. Line 4: public class Test. Line 5: left brace. Line 6, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 7, indented once: left brace. Line 8, indented twice: String string Read, semicolon. Line 9, indented twice: try. Line 10, indented twice: left brace. Line 11, indented thrice: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, data dot t x t, double quotes, right parentheses, right parentheses, semicolon. Line 12: blank. Line 13, indented thrice: string Read, equals, file dot next Line, left parentheses, right parentheses, semicolon. Line 14, indented twice: right brace. Line 15, indented twice: catch, left parentheses, I O Exception i o e, right parentheses. Line 16, indented twice: left brace. Line 17, indented thrice: i o e dot print Stack Trace, left parentheses, right parentheses, semicolon. Line 18, indented twice: right brace. Line 19, indented twice: System dot out dot print l n, left parentheses, double quotes, string read, colon, blank space, double quotes, plus, string Read, right parentheses, semicolon. Line 20, indented twice: 2 forward slashes, line above is line 21. Line 21, indented once: right brace. Line 22: right brace.

Back to Code

Line 1: Test dot java, colon, 21, colon, error, colon, variable string Read might not have been. Line 2: initialized. Line 3: System dot out dot print l n, left parentheses, double quotes, string read, colon, blank space, double quotes, plus, string Read, right parentheses, semicolon. Line 4: a caret sign under s of the string Read variable in Line 3. Line 5: 1 error.

Back to Code

Program code. The code contains the following keywords: new, try, while, int, catch. Line 1: Scanner parse, equals, new Scanner, left parentheses, double quotes, 1 blank space 3.5 blank space 6, double quotes, right parentheses, semicolon. Line 2: try. Line 3: left brace. Line 4, indented once: while, left parentheses, parse dot has Next, left parentheses, right parentheses, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: int number, equals, parse dot next Int, left parentheses, right parentheses, semicolon. Line 7, indented twice: System dot out dot print l n, left parentheses, number, right parentheses, semicolon. Line 8, indented once: right brace. Line 9: right brace. Line 10: catch, left parentheses, Input Mismatch Exception e, right parentheses. Line 11: left brace. Line 12, indented once: System dot out dot print l n, left parentheses, double quotes, In catch block, double quotes, right parentheses, semicolon. Line 13: right brace.

Back to Code

Line 1: 1. Line 2: In catch block.

Back to Code

Program code. The code contains the following keywords: try, new, String, catch. Line 1: try. Line 2: left brace. Line

3, indented once: Scanner file, equals, new Scanner, left parentheses, new File, left parentheses, double quotes, data t x t, double quotes, right parentheses, right parentheses, semicolon. Line 4: blank. Line 5, indented once: String s, equals, file dot next Line, left parentheses, right parentheses, semicolon. Line 6, indented once: System dot out dot print l n, left parentheses, double quotes, Line read is, double quotes, plus, s, right parentheses, semicolon. Line 7: right brace. Line 8: catch, left parentheses, I O Exception i o e, right parentheses. Line 9: left brace. Line 10, indented once: System dot out dot print l n, left parentheses, i o e dot get Message, left parentheses, right parentheses, right parentheses, semicolon. Line 11: right brace.

Back to Code

The output is as follows: data t x t, left parentheses, The system cannot find the file specified, right parentheses.

Back to Code

Program code. The code contains the following keywords: import, public, class, static, void, String, new. Line 1: import java dot i o dot asterisk, semicolon. Line 2: public class Test. Line 3: left brace. Line 4, indented once: public static void main, left parentheses, String, left square bracket, right square bracket, a r g s, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: File Writer f w, equals, new File Writer, left parentheses, double quotes, data dot t x t, double quotes, comma, false, right parentheses, semicolon. Line 7, indented twice: Print Writer p w, equals, new Print Writer, left parentheses, f w, right parentheses, semicolon. Line 8: blank. Line 9, indented twice: p w dot print l n, left parentheses, double quotes, hi, double quotes, right parentheses, semicolon. Line 10, indented twice: p w dot close, left parentheses, right parentheses, semicolon. Line 11, indented once: right brace. Line 12: right brace.

Back to Code

Line 1: Test dot java, colon, 7, colon, error, colon, unreported exception I O Exception, semicolon, must be. Line 2: caught or declared to be thrown. Line 3, indented once: File Writer f w, equals, new File Writer, left parentheses, double quotes, data dot t x t, double quotes, comma, false, right parentheses, semicolon. Line 4: a caret sign under n of new in Line 3. Line 5: 1 error.

Back to Code

first, equals, Mike, ampersand, last, equals, Jones, ampersand, i d, equals, mike 1, ampersand, password, equals, hello.

Back to Code

Line 1: first, colon, Mike. Line 2: last, colon, Jones. Line 3: i d, colon, mike 1. Line 4: password, colon, hello.

Back to Code

The pseudocode contains the following keywords: void, int, if, else. Line 1: void print Hello World N Times, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, n is greater than 0, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: print, double quotes, Hello World, double quotes. Line 6, indented twice: print Hello World N Times, left parentheses, n minus 1, right parentheses. Line 7, indented once: right brace. Line 8, indented once: 2 forward slashes, else do nothing. Line 9: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, int, return. Line 1: public static int multiply Absolute Value By 3, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: return, left parentheses, 3 asterisk Math dot a b s, left parentheses, n, right parentheses, right parentheses, semicolon. Line 4: right brace.

Back to Code

return, left parentheses, expression including a recursive call to the method, right parentheses, semicolon.

Back to Code

factorial, left parentheses, n, right parentheses, equals, n factorial, equals, n asterisk, left parentheses, n minus 1, right parentheses, asterisk, left parentheses, n minus 2, right parentheses, asterisk, left parentheses, n minus 3, right parentheses, asterisk, dot, dot, dot, asterisk, 2, asterisk, 1.

Back to Equation

factorial, left parentheses, 0, right parentheses, equals, 0 factorial, equals, 1.

Back to Equation

factorial, left parentheses, n minus 1, right parentheses, equals, left parentheses, n minus 1, right parentheses, factorial, equals, left parentheses, n minus 1, right parentheses, asterisk, left parentheses, n minus 2, right parentheses, asterisk, left parentheses, n minus 3, right parentheses, asterisk, dot, dot, dot, asterisk, 2, asterisk, 1.

Back to Equation

factorial, left parentheses, n, right parentheses, equals, n asterisk factorial, left parentheses, n minus 1, right parentheses.

Back to Equation

5 factorial, equals, 5 asterisk 4 asterisk 3 asterisk 2 asterisk 1, equals, 120.

Back to Equation

Program code. The code contains the following keywords: public, static, int, return. Line 1: public static int factorial, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, n must be a positive integer. Line 4, indented once: return, left parentheses, n asterisk factorial, left parentheses, n minus 1, right parentheses, right parentheses, semicolon. Line 5: right brace.

Back to Code

Step 1 is as follows. Line 1: r 0, equals, a, percentage sign, b. Line 2: if, left parentheses, r 0 is equal to 0, right parentheses. Line 3, indented once: g c d, left parentheses, a, comma, b, right parentheses, equals, b. Line 4, indented once: stop. Line 5: else. Line 6, indented once: go to step 2. Step 2 is as follows. Line 1: repeat step 1 with b and r 0, instead of a and b.

Back to Code

Step 1 is as follows. Line 1: r 0, equals, 50, percentage sign, 20, equals, 10. Line 2: is 10 equal to 0, question mark, no, go to Step 2. Step 2 is as follows. Line 1: r 0, equals, 20, forward slash, 10, equals, 0. Line 2: is 0 equal to 0, question mark. Line 3, indented once: yes dot g c d, left parentheses, 50, comma, 20, right parentheses, equals, 10. Line 4, indented once: stop.

Back to Code

The formula reads: Combinations, left parentheses, n, comma, p, right parentheses, equals, Combinations, left parentheses, n minus 1, comma, p minus 1, right parentheses, plus, Combinations, left parentheses, n minus 1, comma, p, right parentheses.

Back to Code

Combinations, left parentheses, n, comma, p, right parentheses, equals, n exclamation mark, forward slash, left parentheses, left parentheses, n minus p, right parentheses, exclamation mark, asterisk, p, exclamation mark, right parentheses.

Back to Equation

Combinations, left parentheses, 5, comma, 2, right parentheses, equals, 5 exclamation mark, forward slash, left parentheses, 3, exclamation mark, asterisk, 2, exclamation mark, right parentheses, equals, 10.

Back to Equation

Combinations, left parentheses, n, comma, p, right parentheses, equals, Combinations, left parentheses, n minus 1, comma, p minus 1, right parentheses, plus, Combinations, left parentheses, n minus 1, comma, p, right parentheses.

Back to Equation

n exclamation mark, forward slash, left parentheses, left parentheses, n minus p, right parentheses, exclamation mark, asterisk, p, exclamation mark, right parentheses, equals, left parentheses, n minus 1, right parentheses, exclamation mark, forward slash, left parentheses, left parentheses, n minus p, right parentheses, exclamation mark, asterisk, left parentheses, p minus 1, right parentheses, exclamation mark, right parentheses, plus, left parentheses, n minus 1, right parentheses, exclamation mark, forward slash, left parentheses, left parentheses, n minus 1 minus p, right parentheses, exclamation mark, asterisk, p, exclamation mark, right parentheses.

Back to Equation

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 12. Index: 0. Column 2. Value: 8. Index: 1. Column 3. Value: 10. Index: 2. Column 4. Value: 5. Index: 3. Columns 2 and 3 are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 12. Index: 0. Column 2. Value: 8. Index: 1. Column 3. Value: 10. Index: 2. Column 4. Value: 5. Index: 3. Columns 2, 3 and 4 are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 12. Index: 0. Column 2. Value: 8. Index: 1. Column 3. Value: 10. Index: 2. Column 4. Value: 5. Index: 3. Column 2 is shaded blue. Columns 3 and 4 are shaded red.

Back to Table

System dot out dot print l n, left parentheses, double quotes, sum called with index, double quotes, plus, index, right parentheses, semicolon.

Back to Code

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. The entries in Column 9 are colored red. All the columns are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The

column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. The entries in Column 4 are colored red. Columns 1 to 8 are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. The entries in Column 2 are colored red. Columns 1 to 3 are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. The entries in Column 3 are colored red. Column 3 is shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. The entries in Column 9 are colored red. All the columns are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. The entries in Column 4 are colored red. Columns 1 to 8 are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. The entries in Column 6 are colored

red. Columns 5 to 8 are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72. Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. The entries in Column 7 are colored red. Columns 7 and 8 are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 3. Index: 0. Column 2. Value: 6. Index: 1. Column 3. Value: 7. Index: 2. Column 4. Value: 8. Index: 3. Column 5. Value: 12. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 22. Index: 6. Column 8. Value: 36. Index: 7. Column 9. Value: 45. Index: 8. Column 10. Value: 48. Index: 9. Column 11. Value: 51. Index: 10. Column 12. Value: 53. Index: 11. Column 13. Value: 64. Index: 12. Column 14. Value: 69. Index: 13. Column 15. Value: 72.

Index: 14. Column 16. Value: 89. Index: 15. Column 17. Value: 95. Index: 16. The entries in Column 8 are colored red. Columns 8 is shaded blue.

Back to Table

The method header is as follows: public void recursive T O f H, left parentheses, int num Disks, comma, int from Tower, comma, int to Tower, comma, int use Tower, right parentheses.

Back to Code

recursive T O f H, left parentheses, 5, comma, 0, comma, 2, comma, 1, right parentheses, semicolon.

Back to Code

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 66. Index: 0. Column 2. Value: 54. Index: 1. Column 3. Value: 75. Index: 2. Column 4. Value: 23. Index: 3. Column 5. Value: 15. Index: 4. Column 6. Value: 10. Index: 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: 7.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 23. Index: 0. Column 2. Value: 54. Index: 1. Column 3. Value: 66. Index: 2. Column 4. Value: 75. Index: 3. Column 5.

Value: 15. Index: 4. Column 6. Value: 10. Index: 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: 7. The first 4 columns in the first row are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 23. Index: 0. Column 2. Value: 54. Index: 1. Column 3. Value: 66. Index: 2. Column 4. Value: 75. Index: 3. Column 5. Value: 10. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 80. Index: 6. Column 8. Value: 98. Index: 7. Columns 5 to 8 in row 1 are shaded.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 10. Index: 0. Column 2. Value: 15. Index: 1. Column 3. Value: 23. Index: 2. Column 4. Value: 54. Index: 3. Column 5. Value: 66. Index: 4. Column 6. Value: 75. Index: 5. Column 7. Value: 80. Index: 6. Column 8. Value: 98. Index: 7.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 23. Index: i equals 0. Column 2. Value: 54. Index: 1. Column 3. Value: 66. Index: 2. Column 4. Value: 75. Index: 3. Column 5. Value: 10. Index: j equals 4. Column 6. Value:

15. Index: 5. Column 7. Value: 80. Index: 6. Column 8. Value: 98. Index: 7. Columns 1 to 4 are shaded blue. Columns 5 to 8 are shaded red.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 0. Index: k equals 0. Column 2. Value: 0. Index: 1. Column 3. Value: 0. Index: 2. Column 4. Value: 0. Index: 3. Column 5. Value: 0. Index: j equals 4. Column 6. Value: 0. Index: 5. Column 7. Value: 0. Index: 6. Column 8. Value: 0. Index: 7.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 23. Index: i equals 0. Column 2. Value: 54. Index: k equals 1. Column 3. Value: 66. Index: 2. Column 4. Value: 75. Index: 3. Column 5. Value: 10. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 80. Index: j equals 6. Column 8. Value: 98. Index: 7. Columns 1 to 4 are shaded blue. Columns 5 to 8 are shaded red.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 10. Index: 0. Column 2. Value: 15. Index: 1. Column 3. Value: 0. Index: k equals 2. Column 4. Value: 0. Index: 3.

Column 5. Value: 0. Index: 4. Column 6. Value: 0. Index: 5. Column 7. Value: 0. Index: 6. Column 8. Value: 0. Index: 7. The cells in the first row and the first 2 columns are shaded gray.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 23. Index: 0. Column 2. Value: 54. Index: i equals 1. Column 3. Value: 66. Index: 2. Column 4. Value: 75. Index: 3. Column 5. Value: 10. Index: 4. Column 6. Value: 15. Index: 5. Column 7. Value: 80. Index: j equals 6. Column 8. Value: 98. Index: 7. Columns 1 to 4 are shaded blue. Columns 5 to 8 are shaded red.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 10. Index: 0. Column 2. Value: 15. Index: 1. Column 3. Value: 23. Index: 2. Column 4. Value: 0. Index: k equals 3. Column 5. Value: 0. Index: 4. Column 6. Value: 0. Index: 5. Column 7. Value: 0. Index: 6. Column 8. Value: 0. Index: 7. The cells in the first row and the first 3 columns are shaded gray.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 23.

Index: 0. Column 2. Value: 54. Index: 1. Column 3. Value: 66. Index: 2. Column 4. Value: 75. Index: 3. Column 5. Value: 10. Index: i equals 4. Column 6. Value: 15. Index: 5. Column 7. Value: 80. Index: j equals 6. Column 8. Value: 98. Index: 7. Columns 1 to 4 are shaded blue. Columns 5 to 8 are shaded red.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 10. Index: 0. Column 2. Value: 15. Index: 1. Column 3. Value: 23. Index: 2. Column 4. Value: 54. Index: 3. Column 5. Value: 66. Index: 4. Column 6. Value: 75. Index: 5. Column 7. Value: 0. Index: k equals 6. Column 8. Value: 0. Index: 7. The cells in the first row and the first 6 columns are shaded gray.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 10. Index: 0. Column 2. Value: 15. Index: 1. Column 3. Value: 23. Index: 2. Column 4. Value: 54. Index: 3. Column 5. Value: 66. Index: 4. Column 6. Value: 75. Index: 5. Column 7. Value: 80. Index: 6. Column 8. Value: 98. Index: 7. The cells in the first row are shaded.

Back to Table

Line 1: System dot out dot print l n, left parentheses, double quotes, merge Sort, left parentheses, double quotes, plus, start, plus, double quotes, comma, double quotes, plus, end, plus, double quotes, right parentheses, called, double quotes, right parentheses, semicolon. Line 2: System dot out dot print l n, left parentheses, double quotes, merge, left parentheses, double quotes, plus, start 1, plus, double quotes, comma, double quotes, plus, end 1. Line 3, indented multiple times: plus, double quotes, comma, double quotes, plus, start 2, plus, double quotes, comma, double quotes, plus, end 2, plus, double quotes, right parentheses, called, double quotes, right parentheses, semicolon.

Back to Code

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 66. Index: 0. Column 2. Value: 54. Index: 1. Column 3. Value: 75. Index: 2. Column 4. Value: 23. Index: 3. Column 5. Value: 15. Index: 4. Column 6. Value: 10. Index: 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: 7.

Back to Table

The table has 2 rows labeled Value and Index. The table has 3 groups of columns. The first group of columns is labeled blue and it is marked elements less than pivot, left parentheses, 66, right parentheses. The second group of columns is colored red and it is labeled pivot, left parentheses, 66, right parentheses. The third group of columns is colored brown and it is labeled elements greater than or equal to pivot, left parentheses, 66, right parentheses.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 10. Index: 0. Column 2. Value: 54. Index: 1. Column 3. Value: 23. Index: 2. Column 4. Value: 15. Index: 3. Column 5. Value: 66. Index: j equals 4. Column 6. Value: 75. Index: 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: 7. Columns 1 to 4 are colored blue. Column 5 is colored red. Columns 6 to 8 are colored brown.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 10. Index: 0. Column 2. Value: 15. Index: 1. Column 3. Value: 23. Index: 2. Column 4. Value: 54. Index: 3. Column 5. Value: 66. Index: 4. Column 6. Value: 75. Index: 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: 7. Columns 1 to 4 in row 1 are shaded blue. Columns 6 to 8 in row 1 are shaded red.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 10.

Index: 0. Column 2. Value: 15. Index: 1. Column 3. Value: 23. Index: 2. Column 4. Value: 54. Index: 3. Column 5. Value: 66. Index: 4. Column 6. Value: 75. Index: 5. Column 7. Value: 80. Index: 6. Column 8. Value: 90. Index: 7. Columns 1 to 4 in row 1 are shaded blue. Columns 6 to 8 in row 1 are shaded red.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 66. Index: j equals 0. Column 2. Value: 54. Index: i equals 1. Column 3. Value: 75. Index: 2. Column 4. Value: 23. Index: 3. Column 5. Value: 15. Index: 4. Column 6. Value: 10. Index: 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: 7.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 66. Index: 0. Column 2. Value: 54. Index: j equals 1. Column 3. Value: 75. Index: i equals 2. Column 4. Value: 23. Index: 3. Column 5. Value: 15. Index: 4. Column 6. Value: 10. Index: 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: 7.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 66.

Index: 0. Column 2. Value: 54. Index: j equals 1. Column 3. Value: 75. Index: 2. Column 4. Value: 23. Index: i equals 3. Column 5. Value: 15. Index: 4. Column 6. Value: 10. Index: 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: 7. Columns 1 and 2 are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 66. Index: 0. Column 2. Value: 54. Index: 1. Column 3. Value: 23. Index: j equals 2. Column 4. Value: 75. Index: 3. Column 5. Value: 15. Index: i equals 4. Column 6. Value: 10. Index: 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: 7. Columns 1, 2 and 3 are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 66. Index: 0. Column 2. Value: 54. Index: 1. Column 3. Value: 23. Index: 2. Column 4. Value: 15. Index: j equals 3. Column 5. Value: 75. Index: 4. Column 6. Value: 10. Index: i equals 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: 7. Columns 1 to 4 are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 66. Index: 0. Column 2. Value: 54. Index: 1. Column 3. Value:

23. Index: 2. Column 4. Value: 15. Index: 3. Column 5. Value: 10. Index: j equals 4. Column 6. Value: 75. Index: 5. Column 7. Value: 98. Index: i equals 6. Column 8. Value: 80. Index: 7. Columns 1 to 5 are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 66. Index: 0. Column 2. Value: 54. Index: 1. Column 3. Value: 23. Index: 2. Column 4. Value: 15. Index: 3. Column 5. Value: 10. Index: j equals 4. Column 6. Value: 75. Index: 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: i equals 7. Columns 1 to 5 are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 66. Index: 0. Column 2. Value: 54. Index: 1. Column 3. Value: 23. Index: 2. Column 4. Value: 15. Index: 3. Column 5. Value: 10. Index: j equals 4. Column 6. Value: 75. Index: 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: i equals 7. Columns 1 to 5 are shaded blue.

Back to Table

The table has 2 rows labeled Value and Index. The column entries are as follows. Column 1. Value: 10. Index: 0. Column 2. Value: 54. Index: 1. Column 3. Value: 23. Index: 2. Column 4. Value: 15. Index: 3. Column 5.

Value: 66. Index: j equals 4. Column 6. Value: 75. Index: 5. Column 7. Value: 98. Index: 6. Column 8. Value: 80. Index: 7. Columns 1 to 4 are shaded blue. Column 5 is shaded red. Columns 6 to 8 are shaded brown.

Back to Table

Line 1: System dot out dot print l n, left parentheses, double quotes, quick Sort, left parentheses, double quotes, plus, start, plus, double quotes, comma, double quotes, plus, end, plus, double quotes, right parentheses, called, double quotes, right parentheses, semicolon. Line 2: System dot out dot print l n, left parentheses, double quotes, partition, left parentheses, double quotes, plus, start, plus, double quotes, comma, double quotes, plus, end, plus, double quotes, right parentheses, called, double quotes, right parentheses, semicolon.

Back to Code

Chapter Review of the book. The background image shows a tunnel like path through a forest.

Back to Code

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: public static int foo 1, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, n equals equals 0, right parentheses. Line 4, indented twice: return 0, semicolon. Line 5, indented once: else if, left parentheses, n greater than 0, right parentheses. Line 6, indented twice: return foo 1, left parentheses, n minus 1, right parentheses, semicolon. Line 7, indented once: else. Line 8, indented twice: return foo 1, left parentheses, n plus 1, right parentheses, semicolon. Line 9: right brace.

Back to Code

int i equals foo 1, left parentheses, 0, right parentheses, semicolon.

Back to Code

int i equals foo 1, left parentheses, 4, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: public static int foo 2, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, n is guaranteed to be greater than or equal to 0. Line 4, indented once: if, left parentheses, n less than 10, right parentheses. Line 5, indented twice: return n, semicolon. Line 6, indented once: else. Line 7, indented twice: return foo 2, left parentheses, n minus 10, right parentheses, semicolon. Line 11: right brace.

Back to Code

int i equals foo 2, left parentheses, 7, right parentheses,

semicolon.

Back to Code

int i equals foo 2, left parentheses, 13, right parentheses, semicolon.

Back to Code

int i equals foo 2, left parentheses, 65, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, static, void, String, if. Line 1: public static void foo 3, left parentheses, String s, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, s dot length, left parentheses, right parentheses, greater than 0, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: System dot out dot print, left parentheses, s dot char At, left parentheses, s dot length, left parentheses, right parentheses, minus, 1, right parentheses, right parentheses, semicolon. Line 6, indented twice: foo 3, left parentheses, s dot sub string, left parentheses, 0, comma, s dot length, left parentheses, right parentheses, minus 1, right parentheses, right parentheses, semicolon. Line 7, indented once: right brace. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: public static int foo 4, left parentheses, int n, comma, int p, right parentheses. Line 2: left parentheses. Line 3, indented once: 2 forward slashes, p is guaranteed to be greater than or equal to 0. Line 4, indented once: if, left parentheses, p equals equals 0, right parentheses. Line 5, indented twice: return 1, semicolon. Line 6, indented once: else. Line 7, indented twice: return, left parentheses, n, asterisk, foo 4, left parentheses, n, comma, p minus 1, right parentheses, right parentheses, semicolon. Line 8: right brace.

Back to Code

int i equals foo 4, left parentheses, 6, comma, 0, right parentheses, semicolon.

Back to Code

int i equals foo 4, left parentheses, 5, comma, 1, right parentheses, semicolon.

Back to Code

int i equals foo 4, left parentheses, 4, comma, 3, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, static, int, String, char, if, else, return. Line 1:

public static int foo, left parentheses, String s, comma, char c, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, s dot length, left parentheses, right parentheses, equals equals, 0, right parentheses. Line 4, indented twice: return 0, semicolon. Line 5, indented once: else. Line 6, indented once: left brace. Line 7, indented twice: 2 forward slashes, your code goes here. Line 8, indented once: right brace. Line 9: right brace.

Back to Code

Program code. The code contains the following keywords: public static String foo, left parentheses, String s, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, s dot length, left parentheses, right parentheses, equals, equals, 0, right parentheses. Line 4, indented twice: return, double quotes, even, double quotes, semicolon. Line 5, indented once: else if, left parentheses, s dot length, left parentheses, right parentheses, equals, equals, 1, right parentheses. Line 6, indented twice: return, double quotes, odd, double quotes, semicolon. Line 7, indented once: else. Line 8, indented twice: 2 forward slashes, your code goes here. Line 9: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: public static int foo, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes. n is guaranteed to be greater than or equal to 0. Line 4, indented once: if, left parentheses, n equals equals 0, right parentheses. Line 5, indented twice: return 0, semicolon. Line 6, indented once: else. Line 7, indented once: left brace. Line 8, indented twice: 2 forward slashes, your code goes here. Line 9, indented once: right brace. Line 10: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, boolean, String, if, else, return. Line 1: public static boolean foo, left parentheses, String s, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, blank spaces, right parentheses. 2 forward slashes, base case, hash, 1. Line 4, indented once: else if, left parentheses, blank spaces, right parentheses. 2 forward slashes, base case, hash, 2. Line 5, indented once: else. 2 forward slashes, general case. Line 6, indented twice: return foo, left parentheses, s dot substring, left parentheses, 1, comma, s dot length, left parentheses, right parentheses, right parentheses, right parentheses, semicolon. Line 7: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, int, if, else. Line 1: public static int foo, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, n is guaranteed to be greater than 1. Line 4, indented once: if, left parentheses, n greater than or equal to 1000, right parentheses. 2 forward slashes, base case. Line 5: blank. Line 6, indented once: else. 2 forward slashes, general case. Line 7: right brace.

Back to Code

Program code. The code contains the following keywords: int, public, static, int, if, else, return. Line 1: int p equals foo, left parentheses, 4, right parentheses, semicolon. Line 2: 2 forward slashes, more code here. Line 3: blank. Line 4: public static int foo, left parentheses, int n, right parentheses. Line 5: left brace. Line 6, indented once: int p equals foo, left parentheses, n minus 1, right parentheses, semicolon. Line 7, indented once: if, left parentheses, n equals equals 0, right parentheses. Line 8, indented twice: return 1, semicolon. Line 9, indented once: else. Line 10, indented twice: return, left parentheses, n asterisk p, right parentheses, semicolon. Line 11: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, double, int, if, else, return. Line 1: public static double foo, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, n equals equals 0, right parentheses. Line 4, indented twice: return 1.0, semicolon. Line 5, indented once: else if, left parentheses, n less than 0, right parentheses. Line 6, indented twice: return foo, left parentheses, n minus 1, right parentheses, semicolon. Line 7, indented once: else. Line 8, indented twice: return foo, left parentheses, n plus 1, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, static, boolean, int, if, else, return, true. Line 1: public static boolean foo, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, n is guaranteed to be greater than or equal to 0. Line 4, indented once: if, left parentheses, n equals equals 0, right parentheses. Line 5, indented twice: return true, semicolon. Line 6, indented once: else. Line 7, indented twice: foo, left parentheses, n minus 1, right parentheses, semicolon. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, boolean, int, if, else, return, true. Line 1: public static boolean foo, left parentheses, int n, right parentheses. Line 2: left brace. Line 3, indented once: 2 forward slashes, n is guaranteed to be greater than or equal to 0. Line 3, indented once: if, left parentheses, n equals equals 0, right parentheses. Line 4, indented twice: return true, semicolon. Line 5, indented once: else. Line 6, indented twice: return foo, left parentheses, n, right parentheses, semicolon. Line 7: right brace.

Back to Code

Program code. The code contains the following keywords: public, static, int, return. Line 1: 2 forward slashes, inside main. Line 2: System dot out dot print l n, left parentheses, foo, left parentheses, 5, right parentheses, right parentheses, semicolon. Line 3: 2 forward slashes, more code here. Line 4: blank. Line 5: public static int foo, left parentheses, int n, right parentheses. Line 6: left brace. Line 7, indented once: return, left parentheses, n asterisk foo, left parentheses, n minus 1, right parentheses, right parentheses, semicolon. 2 forward slashes, line 15. Line 8: right brace.

Back to Code

Line 1: Exception in thread, double quotes, main, double quotes, java dot lang dot Stack Overflow Error. Line 2: at Test dot foo, left parentheses, Test dot java, colon, 15, right parentheses. Line 3: at Test dot foo, left parentheses, Test dot java, colon, 15, right parentheses. Line 4: at Test dot foo, left parentheses, Test dot java, colon, 15, right parentheses. Line 5: dot, dot, dot, dot.

Back to Code

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: 2 forward slashes, inside main. Line 2: System dot out dot print l n, left parentheses, foo, left parentheses, 5, right parentheses, right parentheses, semicolon. Line 3: 2 forward slashes, more code here. Line 4: blank. Line 5: public static int foo, left parentheses, int n, right parentheses. Line 6: left brace. Line 7, indented once: if, left parentheses, n equals equals 0, right parentheses. Line 7, indented twice: return foo, left parentheses, 0, right parentheses, semicolon. 2 forward slashes, line 15. Line 8, indented once: else. Line 9, indented twice: return, left parentheses, n asterisk foo, left parentheses, n minus 1, right parentheses, right parentheses, semicolon. Line 10: right brace.

Back to Code

Line 1: Exception in thread, double quotes, main, double quotes, java dot lang, dot, Stack Overflow Error. Line 2: at Test dot foo, left parentheses, Test dot java, colon, 15, right parentheses. Line 3: at Test dot foo, left parentheses, Test dot java, colon, 15, right parentheses. Line 4: at Test dot foo, left parentheses, Test dot java, colon, 15, right parentheses. Line 5: dot, dot, dot, dot.

Back to Code

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: 2 forward slashes, inside main. Line 2: System dot out dot print l n, left parentheses, foo, left parentheses, 5, right parentheses, right parentheses, semicolon. Line 3: 2 forward slashes, more code here. Line 4: blank. Line 5: public static int foo, left parentheses, int n, right parentheses. 2 forward slashes, line 9. Line 6: left brace. Line 7, indented once: if, left parentheses, n, equals, equals, 0, right parentheses. Line 8, indented twice: return 1, semicolon. Line 9, indented once: else. Line 10, indented twice: System dot out dot print l n, left parentheses, n asterisk foo, left parentheses, n minus 1, right parentheses, right parentheses, semicolon. Line 11: right brace. Tab spaces. 2 forward slashes, line 15.

Back to Code

Line 1: Test dot java, colon, 15, colon, error, colon, missing return statement. Line 2: right brace. Tab spaces. 2 forward slashes, line 15. Line 3: a caret sign under the right brace in Line 2. Line 4: 1 error.

Back to Code

Program code. The code contains the following keywords: public, static, int, if, else, return. Line 1: 2 forward slashes, inside main. Line 2: System dot out dot print l n, left parentheses, foo, left parentheses, 5, right parentheses, right parentheses, semicolon. Line 3: 2 forward slashes, more code here. Line 4: blank. Line 5: public static int foo, left parentheses, int n, right parentheses. Line 6: left brace. Line 7, indented once: if, left parentheses, n, equals, equals, 0, right parentheses. Line 8, indented twice: return 1, semicolon. Line 9, indented once: else. Line 10, indented twice: return, left parentheses, foo, left parentheses, n, right parentheses, asterisk, left parentheses, n minus 1, right parentheses, right parentheses, semicolon. Line 11: right brace.

Back to Code

Line 1: Exception in thread, double quotes, main, double quotes, java dot lang dot Stack Overflow Error. Line 2, indented once: at Test dot foo, left parentheses, Test dot java, colon, 15, right parentheses. Line 3, indented once: at Test dot foo, left parentheses, Test dot java, colon, 15, right parentheses. Line 4, indented once: at Test dot foo, left parentheses, Test dot java, colon, 15, right parentheses.

Back to Code

Program code. The code contains the following keywords: public, static, int, String, if, else, return. Line 1: 2 forward slashes, inside main. Line 2: System dot out dot print l n, left parentheses, foo, left parentheses, double quotes, Hello, double quotes, right parentheses, right parentheses, semicolon. 2 forward slashes, line 6. Line 3: 2 forward slashes, more code here. Line 4: public static int foo, left parentheses, String s, right parentheses. 2 forward slashes, line 9. Line 5: left brace. Line 6, indented once: if, left parentheses, s dot length, left parentheses, right parentheses, equals, equals, 0, right parentheses. Line 7, indented twice: return 0, semicolon. Line 8, indented once: else. Line 9, indented twice: return, left parentheses, 1, plus. Line 10, indented thrice: foo, left parentheses, s dot substring, left parentheses, 0, comma, s dot length, left parentheses, right parentheses, minus, 2, right parentheses, right parentheses, right parentheses, semicolon. 2 forward slashes, line 15.

Back to Code

Line 1: Exception in thread, double quotes, main, double quotes, java dot lang dot String Index Out Of Bounds Exception, colon, begin 0, comma. Line 2: end negative 1, comma, length 1. Line 3, indented once: at java dot base, forward slash, java dot lang dot String, dot, check Bounds Begin End, left parentheses, String dot java, colon, 4601, right parentheses. Line 4, indented once: at java dot base, forward slash, java dot lang dot String dot substring, left parentheses, String dot java, colon, 2704, right parentheses. Line 5, indented once: at Test dot foo, left parentheses, Test dot java, colon, 15, right parentheses. Line 6, indented once: at Test dot foo, left parentheses, Test dot java, colon, 15, right parentheses. Line 7, indented once: at Test dot foo, left parentheses, Test dot java, colon, 15, right parentheses. Line 8, indented once: at Test dot main, left parentheses, Test dot java, colon, 6, right parentheses.

Back to Code

Program code. The code contains the following keywords: public, static, int, String, if, else, return, null. Line 1: 2 forward slashes, inside main. Line 2: System dot out dot print l n, left parentheses, foo, left parentheses, double quotes, Hello, double quotes, right parentheses, right parentheses, semicolon. 2 forward slashes, line 6. Line 3: 2 forward slashes, more code here. Line 4: blank. Line 5: public static int foo, left parentheses, String s, right parentheses. 2 forward slashes, line 9. Line 6: left brace. Line 7, indented once: if, left parentheses, s dot length, left parentheses, right parentheses, equals, equals, 0, right parentheses. 2 forward slashes, line 11. Line 8, indented twice: return 0, semicolon. Line 9, indented once: else. Line 10, indented once: left brace. Line 11, indented twice: String temp equals null, semicolon. Line 12, indented twice: if, left parentheses, s dot length, left parentheses, right parentheses, greater than, 1, right parentheses. Line 13, indented thrice: temp, equals, s dot substring, left parentheses, 0, comma, s dot length, left parentheses, right parentheses, minus 1, right parentheses, semicolon. Line 14, indented twice: return, left parentheses, 1 plus foo, left parentheses, temp, right parentheses, right parentheses, semicolon. 2 forward slashes, line 18. Line 15, indented once: right brace. Line 16: right brace.

Back to Code

Line 1: Exception in thread, double quotes, main, double quotes, java dot lang dot Null Pointer Exception, colon.

Line 2: Cannot invoke, double quotes, String dot length, left parentheses, right parentheses, double quotes, because, left angled bracket, parameter 1, right angled bracket, is. Line 3: null. Line 4, indented once: at Test dot foo, left parentheses, Test dot java, colon, 11, right parentheses. Line 5, indented once: at Test dot foo, left parentheses, Test dot java, colon, 18, right parentheses. Line 6, indented once: at Test dot foo, left parentheses, Test dot java, colon, 18, right parentheses. Line 7, indented once: at Test dot foo, left parentheses, Test dot java, colon, 18, right parentheses. Line 8, indented once: at Test dot foo, left parentheses, Test dot java, colon, 18, right parentheses. Line 9, indented once: at Test dot foo, left parentheses, Test dot java, colon, 18, right parentheses. Line 10, indented once: at Test dot main, left parentheses, Test dot java, colon, 6, right parentheses.

Back to Code

Fib, left parentheses, 1, right parentheses, equals, 1. Fib, left parentheses, 2, right parentheses, equals, 1. Fib, left parentheses, n, right parentheses, equals, Fib, left parentheses, n minus 1, right parentheses, plus, Fib, left parentheses, n minus 2, right parentheses, for n greater than or equal to 3.

Back to Code

Line 1: private int data, semicolon. Line 2: private Integer

Node next, semicolon.

Back to Code

Program code. The code contains the following keywords: while, null. Line 1: 2 forward slashes, incorrect ordering of expressions, exclamation mark. Line 2: while, left parentheses, current dot get Data, left parentheses, right parentheses, exclamation mark, equals, value. Line 3, indented once: 2 ampersands, current, exclamation mark, equals, null, right parentheses.

Back to Code

throw new Data Structure Exception, left parentheses, double quotes, Some error message here, double quotes, right parentheses, semicolon.

Back to Code

Line 1: access Modifier data Type method Name, left parentheses, parameter list, right parentheses. Line 2, indented multiple times: throws Data Structure Exception.

Back to Code

tail dot set Next, left parentheses, p n, right parentheses.

Back to Code

The expression reads: left parentheses, back plus 1, right parentheses, percentage sign, QUEUE underscore SIZE.

Back to Code

Left parentheses, back plus 1 minus front, right parentheses, percentage sign, QUEUE underscore SIZE, equals equals, 0.

Back to Code

Left parentheses, back plus 1 minus front, right parentheses, percentage sign, QUEUE underscore SIZE, equals, equals, 0.

Back to Code

public class Player implements Comparable, left angled bracket, Player, right angled bracket.

Back to Code

Line 1: Player Node p n, equals, new, Player Node, left parentheses, p, right parentheses, semicolon. 2 forward slashes, Step 1. Line 2: p n dot set Next, left parentheses, current, right parentheses, semicolon. 2 forward slashes, Step 2. Line 3: current dot get Previous, left parentheses, right parentheses, dot, set Next, left parentheses, p n, right parentheses, semicolon. 2 forward slashes, Step 3. Line 4: p n dot set Previous, left parentheses, current dot get Previous, left parentheses, right parentheses, right parentheses, semicolon. 2 forward slashes, Step 4. Line 5: current dot set Previous, left parentheses, p n, right parentheses, semicolon. 2 forward slashes, Step 5. Line

6: number Of Items, plus, plus, semicolon. 2 forward slashes, Step 6.

Back to Code

Line 1: current dot get Previous, left parentheses, right parentheses, dot, set Next, left parentheses, current dot get Next, left parentheses, right parentheses, right parentheses, semicolon. 2 forward slashes, Step 1. Line 2: current dot get Next, left parentheses, right parentheses, dot, set Previous, left parentheses, current dot get Previous, left parentheses, right parentheses, right parentheses, semicolon. 2 forward slashes, Step 2. Line 3: number Of Items, minus, minus, semicolon. 2 forward slashes, Step 3.

Back to Code

The syntax is as follows: access Modifier class Class Name, left angled bracket, Identifier For Generic Class, right angled bracket.

Back to Code

public class Node, left angled bracket, T, right angled bracket.

Back to Code

private T data, semicolon.

Back to Code

The syntax is as follows: Class Name, left angled bracket, Identifier For Generic Class, right angled bracket.

Back to Code

Generic Linked List, left angled bracket, Integer, right angled bracket, numbers, equals, new, Generic Linked List, left angled bracket, Integer, right angled bracket, left parentheses, right parentheses, semicolon.

Back to Code

rest dot to String, left parentheses, right parentheses.

Back to Code

Program code. The code contains the following keywords: public, void. Line 1: public void foo 1, left parentheses, Player p, comma, Player q, right parentheses. Line 2: left brace. Line 3, indented once: insert, left parentheses, p, right parentheses, semicolon. Line 4, indented once: insert, left parentheses, q, right parentheses, semicolon. Line 5: right brace.

Back to Code

Program code. The code contains the following keywords: public, int, while, null, return. Line 1: public int foo 2, left parentheses, right parentheses. Line 2: left brace. Line 3, indented once: Player Node n d equals head, semicolon. Line 4, indented once: int i equals 0, semicolon. Line 5, indented once: while, left parentheses, n d exclamation mark, equals, null, right parentheses. Line 6, indented once: left brace. Line 7, indented twice: i plus, plus, semicolon. Line 8, indented twice: n d equals n d dot get Next, left parentheses, right parentheses, semicolon. Line 9, indented once: right brace. Line 10, indented once: return i, semicolon. Line 11: right brace.

Back to Code

Program code. The code contains the following keywords: public, boolean, if, null return, true, false, else. Line 1: public boolean foo 3, left parentheses, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, number Of Items, greater than, 0, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: head equals null, semicolon. Line 6, indented twice: number Of Items, equals, 0, semicolon. Line 7, indented twice: return true, semicolon. Line 8, indented once: right brace. Line 9, indented once: else. Line 10, indented twice: return false, semicolon. Line 11: right brace.

Back to Code

Program code. The code contains the following keywords: public, int, while, null, if, return. Line 1: public int foo 4, left parentheses, right parentheses. Line 2: left brace. Line 3, indented once: Player Node n d, equals, head, semicolon. Line 4, indented once: int i equals 0, semicolon. Line 5, indented once: while, left parentheses, n d, exclamation mark, equals, null, right parentheses. Line 6, indented once: left brace. Line 7, indented twice: if, left parentheses, n d dot get Player, left parentheses, right parentheses, dot, get Game, left parentheses, right parentheses, dot, equals, left parentheses, double quotes, Sonic, double quotes, right parentheses, right parentheses. Line 8, indented thrice: i plus, plus, semicolon. Line 9, indented twice: n d, equals, n d dot get Next, left parentheses, right parentheses, semicolon. Line 10, indented once: right brace. Line 11, indented once: return i, semicolon. Line 12: right brace.

Back to Code

Program code. The code contains the following keywords: public, boolean, int, while, null, if, return, true, false. Line 1: public boolean foo 5, left parentheses, int i, right parentheses. Line 2: left brace. Line 3, indented once: Player Node n d, equals, head, semicolon. Line 4, indented once: while, left parentheses, n d, exclamation mark, equals, null, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: if, left parentheses, n d dot get Player, left parentheses, right parentheses, dot, get I D, left parentheses, right parentheses, equals, equals, i, right parentheses. Line 7, indented thrice: return true, semicolon. Line 8, indented twice: n d, equals, n d dot get Next, left parentheses, right parentheses, semicolon. Line 9, indented once: right brace. Line 10, indented once: return false, semicolon. Line 11: right brace.

Back to Code

Program code. The code contains the following keywords: public, void, while, null, if. Line 1: public void foo 6, left parentheses, right parentheses. Line 2: left brace. Line 3, indented once: Player Node n d, equals, head, semicolon. Line 4, indented once: while, left parentheses, n d, exclamation mark, equals, null, right parentheses. Line 5, indented once: left brace. Line 6, indented twice: if, left parentheses, n d dot get Player, left parentheses, right parentheses, dot, get Game, left parentheses, right parentheses, dot, equals, left parentheses, double quotes, Diablo, double quotes, right parentheses, right parentheses. Line 7, indented thrice: System dot out dot print l n, left parentheses, n d dot get Player, left parentheses, right parentheses, dot, to String, left parentheses, right parentheses, right parentheses, semicolon. Line 8, indented twice: n d, equals, n d dot get Next, left parentheses, right parentheses, semicolon. Line 9, indented once: right brace. Line 10: right brace.

Back to Code

Program code. The code contains the following keywords: public, void, if, else, new. Line 1: public void foo 7, left parentheses, Player p, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, number Of Items, equals, equals, 0, right parentheses. Line 4, indented twice: System dot out dot print l n, left parentheses, double quotes, Do nothing, double quotes, right parentheses, semicolon. Line 5, indented once: else. Line 6, indented once: left brace. Line 7, indented twice: Player Node p n, equals, new, Player Node, left parentheses, p, right parentheses, semicolon. Line 8, indented twice: p n dot set Next, left parentheses, head dot get Next, left parentheses, right parentheses, right parentheses, semicolon. Line 9, indented twice: head dot set Next, left parentheses, p n, right parentheses, semicolon. Line 10, indented twice: number Of Items, plus, plus, semicolon. Line 11, indented once: right brace. Line 12, indented once: right brace.

Back to Code

Program code. The code contains the following keywords: public, boolean, if, else, true, false, return. Line 1: public boolean foo 8, left parentheses, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, number Of Items, less than or equal to, 2, right parentheses. Line 4, indented twice: return false, semicolon. Line 5, indented once: else. Line 6, indented once: left brace. Line 7, indented twice: head dot set Next, left parentheses, left parentheses, head dot get Next, left parentheses, right parentheses, right parentheses, dot, get Next, left parentheses, right parentheses, right parentheses, semicolon. Line 8, indented twice: number Of Items, minus, minus, semicolon. Line 9, indented twice: return true, semicolon. Line 10, indented once: right brace. Line 11: right brace.

Back to Code

Program code. The code contains the following keywords: public, class, private, char. Line 1: public class L L Node. Line 2: left brace. Line 3, indented once: private char grade, semicolon. Line 4, indented once: private L L Node next, semicolon. Line 5, indented twice: 2 forward slashes, constructors and methods here. Line 6: right brace.

Back to Code

2 forward slashes, your code goes here.

Back to Code

2 forward slashes, your code goes here.

Back to Code

2 forward slashes, your code goes here.

Back to Code

2 forward slashes, your code goes here.

Back to Code

Program code. The code contains the following keywords: public, class, private. Line 1: public class Differential Linked List. Line 2: left brace, tab space, private L L Node head, semicolon. Line 3, indented once: 2 forward slashes, there is no instance variable for. Line 4, indented once: 2 forward slashes, the number of items in the list. Line 5: blank. Line 6, indented once: 2 forward slashes, constructors and methods here. Line 7: right brace.

Back to Code

2 forward slashes, your code goes here.

Back to Code

2 forward slashes, your code goes here.

Back to Code

2 forward slashes, your code goes here.

Back to Code

public Player retrieve Me, left parentheses, int index, right parentheses.

Back to Code

2 forward slashes, your code goes here.

Back to Code

head dot set Next, left parentheses, head, right parentheses, semicolon.

Back to Code

Program code. The code contains the following keywords: public, int, return. Line 1: public int get Head I D, left parentheses, right parentheses. Line 2: left brace. Line 3, indented once: return head dot get I D, left parentheses, right parentheses, semicolon. Line 4: right brace.

Back to Code

Program code. The code contains the following keywords: public, boolean, if, else, true, false, return. Line 1: public boolean push, left parentheses, Player p, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, exclamation mark, is Full, left parentheses, right parentheses, right parentheses. 2 forward slashes, is there room to insert, question mark. Line 4, indented once: left brace. Line 5, indented twice: stack, left parentheses, top, right parentheses, equals, p, semicolon. Line 6, indented twice: return true, semicolon. Line 7, indented once: right brace. Line 8, indented once: else. Line 9, indented twice: return false, semicolon. Line 10: right brace.

Back to Code

Program code. The code contains the following keywords: public, if, else, return, throw, new. Line 1: public Player pop, left parentheses, right parentheses, throws Data Structure Exception. Line 2: left brace. Line 3, indented once: if, left parentheses, exclamation mark, is Empty, left parentheses, right parentheses, right parentheses. 2 forward slashes, is there an item to delete, question mark. Line 4, indented twice: return, left parentheses, stack, left square bracket, top, right square bracket, right parentheses, semicolon. Line 5, indented once: else. Line 6, indented twice: throw new Data Structure Exception. Line 7, indented multiple times: left parentheses, double quotes, Stack empty, colon, cannot pop, double quotes, right parentheses, semicolon. Line 8: right brace.

Back to Code

Program code. The code contains the following keywords: public, boolean, if, else, true, false, return. Line 1: public boolean enqueue, left parentheses, Player p, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, exclamation mark, is Full, left parentheses, right parentheses, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: queue, left square bracket, left parentheses, back, plus, 1, right parentheses, percentage sign, QUEUE underscore SIZE, right square bracket, equals, p, semicolon. Line 6, indented twice: back, equals, left parentheses, back, plus, 1, right parentheses, percentage sign, QUEUE underscore SIZE, semicolon. Line 7, indented twice: return true, semicolon. Line 8, indented once: right brace. Line 9, indented once: else. Line 10, indented twice: return false, semicolon. Line 11: right brace. Program code. The code contains the following keywords: public, boolean, if, else, true, false, return. Line 1: public boolean enqueue, left parentheses, Player p, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, exclamation mark, is Full, left parentheses, right parentheses, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: queue, left square bracket, left parentheses, back, plus, 1, right parentheses, percentage sign, QUEUE underscore SIZE, right square bracket, equals, p, semicolon. Line 6, indented twice: back, equals, left parentheses, back, plus, 1, right parentheses, percentage sign, QUEUE underscore SIZE, semicolon. Line 7, indented twice: return true, semicolon. Line 8, indented once: right brace. Line 9, indented once: else. Line 10, indented twice: return false, semicolon. Line 11: right brace. Program code. The code contains the following keywords: public, boolean, if, else, true, false, return. Line 1: public boolean enqueue, left parentheses, Player p, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, exclamation mark, is Full, left parentheses, right parentheses, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: queue, left square bracket, left parentheses, back, plus, 1, right parentheses, percentage sign, QUEUE underscore SIZE, right square bracket, equals, p, semicolon. Line 6, indented twice: back, equals, left parentheses, back, plus, 1, right parentheses, percentage sign, QUEUE underscore SIZE, semicolon. Line 7, indented twice: return true, semicolon. Line 8, indented once: right brace. Line 9, indented once: else. Line 10, indented twice: return false, semicolon. Line 11: right brace. Program code. The code contains the following keywords: public, boolean, if, else, true, false, return. Line 1: public boolean enqueue, left parentheses, Player p, right parentheses. Line 2: left brace. Line 3, indented once: if, left parentheses, exclamation mark, is Full, left parentheses, right parentheses, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: queue, left square bracket, left parentheses, back, plus, 1, right parentheses, percentage sign, QUEUE underscore SIZE, right square bracket, equals, p, semicolon. Line 6, indented twice: back, equals, left parentheses, back, plus, 1, right parentheses, percentage sign, QUEUE underscore SIZE, semicolon. Line 7, indented twice: return true, semicolon. Line 8, indented once: right brace. Line 9, indented once: else. Line 10, indented twice: return false, semicolon. Line 11: right brace.

Back to Code

Program code. The code contains the following keywords: public, throws, if, else, return, throw, new. Line 1: public Player dequeue, left parentheses, right parentheses, throws Data Structure Exception. Line 2: left brace. Line 3, indented once: if, left parentheses, exclamation mark, is Empty, left parentheses, right parentheses, right parentheses. Line 4, indented once: left brace. Line 5, indented twice: front, equals, left parentheses, front plus 1, right parentheses, percentage sign, QUEUE underscore SIZE, semicolon. Line 6, indented twice: number Of Items, minus, minus, semicolon. Line 7, indented twice: return queue, left square bracket, left parentheses, front minus 1, right parentheses, percentage sign, QUEUE underscore SIZE, right square bracket, semicolon. Line 8, indented once: right brace. Line 9, indented once: else. Line 10, indented twice: throw new Data Structure Exception. Line 11, indented multiple times: left parentheses, double quotes, Queue empty, colon, cannot dequeue, double quotes, right parentheses, semicolon. Line 12: right brace.

Back to Code

Line 1: Player Node p n, equals, new, Player Node, left parentheses, right parentheses, semicolon. Line 2: Player p, equals, p n, dot, get Player, left parentheses, right parentheses, semicolon. Line 3: p dot set I D, left parentheses, 10, right parentheses, semicolon. 2 forward slashes, line 10.

Back to Code

Line 1: Exception in thread, double quotes, main, double quotes, java dot lang dot Null Pointer Exception, colon. Line 2: Cannot invoke, double quotes, Player dot set I D, left parentheses, int, right parentheses, double quotes, because, double quotes, left angled bracket, local 2, right angled bracket, double quotes, is null. Line 3, indented multiple times: at Test dot main, left parentheses, Test dot java, colon, 10, right parentheses.

Back to Code

Line 1: Player p, equals, new Player, left parentheses, 5, comma, double quotes, Ajay, double quotes, comma, double quotes, Mario, double quotes, right parentheses, semicolon. Line 2: Player Linked List p l l, equals, new Player Linked List, left parentheses, right parentheses, semicolon. Line 3: p l l dot insert, left parentheses, p, right parentheses, semicolon. Line 4: Player Node temp, equals, p l l dot get Head, left parentheses, right parentheses, semicolon. 2 forward slashes, line 10. Line 5: System dot out dot print l n, left parentheses, double quotes, head is, double quotes, plus, temp dot to String, left parentheses, right parentheses, right parentheses, semicolon.

Back to Code

Line 1: Test dot java, colon, 10, colon, error, colon, cannot find symbol. Line 2, indented once: Player Node temp, equals, p l l dot get Head, left parentheses, right parentheses, semicolon. 2 forward slashes, line 10. Line 3: a caret sign under the dot folling p l l in Line 2. Line 3, indented once: symbol, colon, method get Head, left parentheses, right parentheses. Line 4, indented once: location, colon, variable p l l of type Player Linked List. Line 5: 1 error.

Back to Code

Line 1: Player, p, equals, new, Player, left parentheses, 5, comma, double quotes, Ajay, double quotes, comma, double quotes, Mario, double quotes, right parentheses, semicolon. Line 2: Player Linked List p l l, equals, new Player Linked List, left parentheses, right parentheses, semicolon. Line 3: p l l dot insert, left parentheses, p, right parentheses, semicolon. Line 4: if, left parentheses, p l l dot delete, left parentheses, 5, right parentheses, right parentheses. 2 forward slashes, line 10. Line 5, indented once: System dot out dot print l n, left parentheses, double quotes, Successful deletion, double quotes, right parentheses, semicolon.

Back to Code

Line 1: Test dot java, colon, 10, colon, error, colon, incompatible types, colon, Player cannot be. Line 2: converted to boolean. Line 3, indented once: if, left parentheses, p l l dot delete, left parentheses, 5, right parentheses, right parentheses. 2 forward slashes, line 10. Line 4: A caret sign uner the left parentheses to the left of 5. Line 5: 1 error.

Back to Code

Line 1: Player Node current, equals, head, semicolon. 2 forward slashes, line 9. Line 2: while, left parentheses, current dot get Player, left parentheses, right parentheses, dot, get I D, left parentheses, right parentheses, exclamation mark, equals, 99, right parentheses. Line 3, indented once: current, equals, current dot get Next, left parentheses, right parentheses, semicolon. Line 4: 2 forward slashes, more code here but no problem.

Back to Code

Line 1: Exception in thread, double quotes, main, double quotes, java dot lang dot Null Pointer Exception, colon. Line 2: Cannot invoke, double quotes, Player Node dot get Player, left parentheses, right parentheses, double quotes, because, double quotes, left angled bracket, local 1, right angled bracket, double quotes, is null. Line 3, indented multiple times: at Player Linked List dot foo, left parentheses, Player Linked List dot java, colon, 10, right parentheses. Line 4, indented multiple times: at Test dot main, left parentheses, Test dot java, colon, 24, right parentheses.

Back to Code

The value of L as a function of n if there are n nodes in a binary tree whose deepest level is L. n equals total number of nodes equals sum of nodes for all levels equals 1 plus 2 to the power of 1 plus 2 to the power of 2 plus 2 to the power of 3 plus 2 to the power of 4 plus 2 to the power of 5 plus and so on until 2 to the power of L.

Back to Equation

A simple proof for the formula. n equals equals 1 plus 2 to the power of 1 plus 2 to the power of 2 plus 2 to the power of 3 plus 2 to the power of 4 plus 2 to the power of 5 plus and so on until 2 to the power of L.

Back to Equation

Multiplying each side of the formula by 2. 2 n, 2 plus 2 to the power of 2 plus 2 to the power of 2 plus 2 to the power of 3 plus 2 to the power of 4 plus 2 to the power of 5 plus and so on plus 2 the power of L plus 2 to the power of L plus 1 the whole.

Back to Equation

Subtracting n from 2 n in the formula. 2 n minus n equals 2 to the power of L plus 1 the whole minus 1. Thus, n equals 2 to the power of L plus 1 the whole, or n plus 1 equals 2 the power of L plus 1 the whole.

Back to Equation

Taking the log of both sides of the equation. Log of n plus 1 the whole equals L plus 1, or L equals log of n plus 1 the whole minus 1.

Back to Equation

Method header. Program code. In the code, the words in the variable names are merged. Line 1: private, String, pre Order, left parenthesis, Integer Node, n d, right parenthesis.

Back to Code

Code in question 13 in section 15.10.2. Program code. In the code, the words in the variable names are merged.

Line 1: n d, dot, set Right, left parenthesis, n d, dot, get Right, left parenthesis, right parenthesis, dot, get Right, left parenthesis, right parenthesis, right parenthesis, semicolon.

Back to Code

Code in question 14 in section 15.10.2. Program code. In the code, the words in the variable names are merged. Line 1: Integer Node, temp, equals, n d, dot, get Left, left parenthesis, right parenthesis, dot, get Right, left parenthesis, right parenthesis, semicolon. Line 2: n d, dot, get Left, left parenthesis, right parenthesis, dot, set Right, left parenthesis, temp, dot, get Right, left parenthesis, right parenthesis, right parenthesis, semicolon.

Back to Code

The pre order and in order traversals in a binary tree. In order: 55, 17, 74, 100, 54, 22, 30, 8, 12. Pre order: 100, 17, 55, 74, 22, 54, 8, 30, 12.

Back to Code

The traversals of a binary tree. Pre order: 88, 44, 60, 25, 10, 33, 40, 100. In order: 60, 44, 10, 25, 88, 33, 100, 40. Post order: 60, 10, 25s, 44, 100, 40, 33, 88.

Back to Code

A binary tree. The root node is 50. The node 50 has the child nodes, 30 and 70. The node 30 has the child nodes, 20 and 40. The node 70 has the child node, 90.

Back to Figure

A binary tree. The root node is 10. The node 10 has the child node, 25. The node 25 has the child node, 50. The node 50 has the child node, 75.

Back to Figure

Before and After picture of a traversal in a binary tree. The before picture has the following nodes. The root node is 10. It has the child node, 25. The node 25 has the child node, 50. The node 50 has the child node, 75. The after picture has the following nodes. The root node is 10. It has the child node, 25. The node 25 has the child node, 50.

Back to Figure

Before and After picture of a traversal in a binary tree. The before picture has the following nodes. The root node is 10. It has the child node, 25. The node 25 has the child node, 50. The node 50 has the child node, 75. The after picture has the following nodes. The root node is 10. It has the child node, 25. The node 25 has the child node, .

Back to Figure

Code in question 24 of section 15.10.3. Program code. In the code, the words in the variable names are merged. Line 1: public, boolean, is Empty, left parenthesis, right parenthesis. Line 2: Left brace. Line 3, indented once: Forward slash, forward slash, Your code goes here. Line 4: Right brace.

Back to Code

Code in question 25 of section 15.10.3. Program code. In the code, the words in the variable names are merged. Line 1: public, boolean, has Exactly One, left parenthesis, right parenthesis. Line 2: Left brace. Line 3, indented once: Forward slash, forward slash, Your code goes here. Line 4: Right brace.

Back to Code

Code in question 26 of section 15.10.3. Program code. In the code, the words in the variable names are merged. Line 1: public, boolean, root Has 2 Children, left parenthesis, right parenthesis. Line 2: Left brace. Line 3, indented once: Forward slash, forward slash, Your code goes here. Line 4: Right brace.

Back to Code

Code in question 27 of section 15.10.3. Program code. In the code, the words in the variable names are merged. Line 1: public, boolean, stores At Least 3 Values, left parenthesis, right parenthesis. Line 2: Left brace. Line 3, indented once: Forward slash, forward slash, Your code goes here. Line 4: Right brace.

Back to Code

Code in question 28 of section 15.10.3. Program code. In the code, the words in the variable names are merged. Line 1: public, i n t, number Of Odd Values, left parenthesis, right parenthesis. Line 2: Left brace. Line 3, indented once: Forward slash, forward slash, Your code goes here. Line 4: Right brace. Line 5: private, i n t, number Of Odd Values, left parenthesis, Integer Node, n d, right parenthesis. Line 6: Left brace. Line 7, indented once: Your code goes here. Line 8: Right brace.

Back to Code

Code in question 29 of section 15.10.4. Program code. In the code, the words in the variable names are merged. Line 1: public, String, pre Order, left parenthesis, right parenthesis. Line 2: Left brace. Line 3, indented once: return, pre Order, left parenthesis, root, right parenthesis, semicolon. Line 4: Right brace. Line 5: private, String, pre Order, left parenthesis, Integer Node, n d, right parenthesis. Line 6: Left brace. Line 7, indented once: String, result, equals, n d, dot, get Data, left parenthesis, right parenthesis, plus, open quotes, space, close quotes, semicolon. Line 8, indented once: result, plus, equals, pre Order, left parenthesis, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 9, indented once: result, plus, equals, pre Order, left parenthesis, n d, dot, get Right, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 10, indented once: return, result, semicolon. Line 11: Right brace.

Back to Code

Code in question 30 of section 15.10.4. Program code. In the code, the words in the variable names are merged. Line 1: public, void, attach Left, left parenthesis, Binary Tree, b t, right parenthesis. Line 2: Left brace. Line 3, indented once: root, equals, root, dot, get Left, left parenthesis, b t, dot, root, right parenthesis, semicolon. Line 4: Right brace.

Back to Code

Code in question 31 of section 15.10.4. Program code. In the code, the words in the variable names are merged. Line 1: public, boolean, delete, left parenthesis, i n t, value, right parenthesis. Line 2: Left brace. Line 3, indented once: return, delete, left parenthesis, value, comma, root, right parenthesis, not, equals, null, semicolon. Line 4: Right brace.

Back to Code

Code in question 32 of section 15.10.4. Program code. In the code, the words in the variable names are merged. Line 1: if, left parenthesis, start, less than or equals, end, right parenthesis. Text: to. Line 1: if, left parenthesis, start, less than, end, right parenthesis.

Back to Code

Code in question 32 of section 15.10.4. Program code. In the code, the words in the variable names are merged. Line 1: if, left parenthesis, start, less than, end, right parenthesis.

Back to Code

Traversals in a binary tree. Pre order: 36, 17, 20, 50, 66. In order: 17, 20, 36, 50, 66. Post order: 20, 17, 66, 50, 36.

Back to Code

Code in question 33 of section 15.10.4. Program code. In the code, the words in the variable names are merged. Line 1: n d, dot, set Left, left parenthesis, rebalance, left parenthesis, a r r, comma, start, comma, middle, minus 1, right parenthesis, right parenthesis, semicolon. Line 2: n d, dot, set Right, left parenthesis, rebalance, left parenthesis, a r r, comma, middle, plus, 1, comma, end, right parenthesis, right parenthesis, semicolon.

Back to Code

Code in question 33 of section 15.10.4. Program code. In the code, the words in the variable names are merged. Line 1: rebalance, left parenthesis, a r r, comma, start, comma, middle, minus, 1, right parenthesis, semicolon. Line 2: rebalance, left parenthesis, a r r, comma, middle, plus, 1, comma, end, right parenthesis, semicolon.

Back to Code

Output as listed in question 33 of section 15.10.4. Pre order: 36. In order: 36. Post order: 36.

Back to Code

Code in question 34 of section 15.10.5. Program code. In the code, the words in the variable names are merged. Line 1: public, void, pre Order 2, left parenthesis, right parenthesis. Line 2: Left brace. Line 3, indented once: pre Order 2, left parenthesis, root, right parenthesis, semicolon. Line 4: Right brace. Line 5: private, void, pre Order 2, left parenthesis, Integer Node, n d, right parenthesis. Line 6: Left brace. Line 7, indented once: System, dot, out, dot, print, left parenthesis, n d, dot, get Data, left parenthesis, right parenthesis, plus, open quotes, space, close quotes, right parenthesis, semicolon. Line 8, indented once: pre Order 2, left parenthesis, n d, dot, get Left, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 9, indented once: pre Order 2, left parenthesis, n d, dot, get Right, left parenthesis, right parenthesis, right parenthesis, semicolon.

Back to Code

The exception that is raised by the code in question 34 in section 15.10.5. A Null Pointer Exception is raised. The get Data function cannot be invoked because parameter 1 is null in line number 125 of the code.

Back to Code

Code in question 35 of section 15.10.5. Program code. In the code, the words in the variable names are merged. Line 1: root, dot, set Left, left parenthesis, b t, dot, root, right parenthesis, semicolon.

Back to Code

Code in question 35 of section 15.10.5. Program code. In the code, the words in the variable names are merged. Line 1: root, dot, set Left, left parenthesis, b t, right parenthesis, semicolon.

Back to Code

Error that is raised by the code in question 35 in section 15.10.5. The error is in line number 24 of the Binary Tree dot java program. The error is of incompatible types. Binary Tree cannot be converted to Integer Node in the code, root, dot set Left, left parenthesis, b t, right parenthesis, semicolon.

Back to Code

Code in question 36 of section 15.10.5. Program code. In the code, the words in the variable names are merged. Line 1: public, void, insert, left parenthesis, i n t, value, right parenthesis. Line 2: Left brace. Line 3, indented once: insert, left parenthesis, value, comma, root, right parenthesis, semicolon. Line 4: Right brace.

Back to Code

The exception that is raised by the code in question 36 in section 15.10.5. A Null Pointer Exception is raised. The get Data function cannot be invoked because parameter 2 is null in lines 32, 26, and 11.

Back to Code

A binary tree. The root node is question mark colon 4. The root node has directed edges 0 and 1. Edge 0 points to the node, question mark, colon, 2. Edge 1 points to the node, A, colon, 2. The node question mark, colon, 2 has two directed edges, 0 and 1. Edge 0 points to the node, J, colon, 1. Edge 1 points to the node, V, colon, 1.

Back to Figure

Header pattern for G U I applications. Program code. In the code, the words in the variable names are merged. Line 1: public, class, Class Name, extends, J Frame.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: public, Image Icon, left parenthesis, String, filename, right parenthesis.

Back to Code

The code pattern to use for adding event handling to the Shell G U I Application class. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, import statements as needed. Line 2: public, class, Class Name 1, extends, J Frame. Line 3: Left brace. Line 4, indented once: forward slash, forward slash, declare components and other instance variables. Line 5, indented once: Forward slash, forward slash, constructor. Line 6, indented once: public, Class Name 1, left parenthesis, right parenthesis. Line 7, indented once: Left brace. Line 8, indented twice: Forward slash, forward slash, call J Frame constructor. Line 9, indented twice: forward slash, forward slash, get content pane. Line 10, indented twice: forward slash, forward slash, set the layout manager. Line 11, indented twice: forward slash, forward slash, instantiate components and other instance variables. Line 12, indented twice: forward slash, forward slash, add components to the content pane. Line 13, indented twice: forward slash, forward slash, declare and instantiate event handler objects. Line 14, indented twice: forward slash, forward slash, register event handlers on components. Line 15, indented twice: forward slash, forward slash, set window size. Line 16, indented twice: forward slash, forward slash, make window visible. Line 17, indented once: Right brace. Line 18, indented once: Forward slash, forward slash, private event handler class. Line 19, indented once: private, class, Event Handler Name, implements, Listener Name. Line 20, indented once: Left brace. Line 21, indented twice: forward slash, forward slash, implement the methods of the listener interface. Line 22, indented twice:

forward slash, forward slash, to process the events. Line 23, indented once: Right brace. Line 24, indented once. public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 25, indented once. Left brace. Line 26, indented twice: forward slash, forward slash, instantiate application object. Line 27, indented once: Right brace. Line 28: Right brace.

Back to Code

A compile error. abstract method, action Performed, left parenthesis, Action Event, right parenthesis, in Action Listener. private class Text Field Handler implements Action Listener. 1 error. The error is indicated next to the keyword, class.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: public, String, left parenthesis, char, left square brace, right square brace, char Array, right parenthesis.

Back to Code

Bulk import statement. Program code. In the code, the words in the variable names are merged. Line 1: import, package Name, dot, asterisk, semicolon.

Back to Code

Program code. In the code, the words in the variable names are merged. this, dot, add Mouse Listener, left parenthesis, m h, right parenthesis, semicolon.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: public, class, Controller. Line 2: Left brace. Line 3, indented once: private, Model Interface, model, semicolon, forward slash, forward slash, Model Interface is an interface. Line 4, indented once: private, View, view, semicolon. Line 5, indented once: dot, dot, dot.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: public, class, Model, implements, Model Interface. Line 2: Left brace. Line 3, indented once: forward slash, forward slash, Implements Model Interface methods. Line 4, indented once: dot, dot, dot.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, a few asterisks, Task 1: declare a J Panel named top. Line 2: Forward slash, forward slash, also declare three J Button instance variables. Line 3: Forward slash, forward slash, that will be added to the J Panel top. Line 4:

Forward slash, forward slash, these buttons will determine the grid size of the game, colon. Line 5: Forward slash, forward slash, 3-by-3, 4-by-4, or 5-by-5. Line 6: Forward slash, forward slash, task 1 ends here.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, a few asterisks, Task 2: student code starts here. Line 2: Forward slash, forward slash, instantiate the Border Layout manager b l. Line 3: Forward slash, forward slash, set the layout manager of the content pane contents to b l. Line 4: game, equals, new, Game, left parenthesis, 3, right parenthesis, semicolon, forward slash, forward slash, instantiating the Game object. Line 5: Forward slash, forward slash, add game to the center of the content pane. Line 6: Forward slash, forward slash, task 2 ends here.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, a few asterisks, Task 3: Student code restarts here. Line 2: Forward slash, forward slash, instantiate the J Panel component named top. Line 3: Forward slash, forward slash, set the layout of top to a 1-by-3 grid. Line 4: Forward slash, forward slash, instantiate the J Buttons that determine the grid size. Line 5: Forward slash, forward slash, add the buttons to J Panel top. Line 6: Forward slash, forward slash, add J Panel top to the content pane as its north component. Line 7: Forward slash, forward slash, task 3 ends here.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, a few asterisks, Task 4: Student code restarts here. Line 2: Forward slash, forward slash, create a private inner class that implements Action Listener. Line 3: Forward slash, forward slash, your method should identify which of the 3 buttons was the. Line 4: Forward slash, forward slash, source of the event. Line 5: Forward slash, forward slash, depending on which button was pressed, comma. Line 6: Forward slash, forward slash, call the set Up Game method of the Game class. Line 7: Forward slash, forward slash, with arguments 3, 4, or 5. Line 8: Forward slash, forward slash, the A P I of that method is: Line 9: Forward slash, forward slash, public, void, set Up Game, left parenthesis, i n t, n Sides, right parenthesis. Line 10: Forward slash, forward slash, task 4 ends here.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, a few asterisks, Task 5: Student code restarts here. Line 2: Forward slash, forward slash, Note: search for and complete Task 4 before performing this task. Line 3: Forward slash, forward slash, declare and instantiate an Action Listener. Line 4: Forward slash, forward slash, register the listener on the 3 buttons. Line 5: Forward slash, forward slash, that you declared in Task 1. Line 6: Forward slash, forward slash, task 5 ends here.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: Grid Layout, g l, equals, new, Grid Layout, left parenthesis, 6, comma, 4, right parenthesis, semicolon.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: Border Layout, b l, equals, new, Border Layout, left parenthesis, 7, comma, 3, right parenthesis, semicolon.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: contents, dot, add, left parenthesis, button, comma, Border Layout, dot, NORTH, right parenthesis, semicolon.

Back to Code

Code to consider for questions 16 to 21. Program code. In the code, the words in the variable names are merged.

Line 1: import, java x, dot, swing, dot, asterisk, semicolon. Line 2: import, java, dot, a w t, dot, asterisk, semicolon. Line 3: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 4: public, class, Game, extends, J Frame. Line 5: Left brace. Line 6, indented once: private, J Button, b 1, comma, b 2, comma, b 3, comma, b 4, semicolon. Line 7, indented once: private, Container, contents, semicolon. Line 8, indented once: public, Game, left parenthesis, right parenthesis. Line 9, indented once: Left brace. Line 10, indented twice: super, left parenthesis, open quotes, Play this game, close quotes, right parenthesis, semicolon. Line 11, indented twice: contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 12, indented twice: contents, dot, set Layout, left parenthesis, new, Grid Layout, left parenthesis, 2, comma, 2, right parenthesis, right parenthesis, semicolon. Line 13, indented twice: b 1, equals, new, J Button, left parenthesis, open quotes, Button 1, close quotes, right parenthesis, semicolon. Line 14, indented twice: b 2, equals, new, J Button, left parenthesis, open quotes, Button 2, close quotes, right parenthesis, semicolon. Line 15, indented twice: b 3, equals, new, J Button, left parenthesis, open quotes, Button 3, close quotes, right parenthesis, semicolon. Line 16, indented twice: b 4, equals, new, J Button, left parenthesis, open quotes, Button 4, close quotes, right parenthesis, semicolon. Line 17, indented once: contents, dot, add, left parenthesis, b 1, right parenthesis, semicolon. Line 18, indented twice: contents, dot, add, left parenthesis, b 2, right parenthesis, semicolon. Line 19, indented twice: contents, dot, add, left parenthesis, b 3, right parenthesis, semicolon. Line 20, indented twice: My Handler, m h, equals, new, My Handler, left parenthesis, right parenthesis, semicolon. Line 21, indented twice: b 1, dot, add Action Listener, left parenthesis, m h, right parenthesis, semicolon. Line 22, indented twice: b 2, dot, add Action Listener, left parenthesis, m h, right parenthesis, semicolon. Line 23, indented twice: set Size, left parenthesis, 400, comma, 400, right parenthesis, semicolon. Line 24, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 25, indented once: right brace. Line 26, indented once: private, class, My Handler, implements, Action Listener. Line 27, indented once: Left brace. Line 28, indented twice: public, void, action Performed, left parenthesis, Action Event, a e, right parenthesis. Line 29, indented twice: Left brace. Line 30, indented 3 times: System, dot, out, dot, print l n, left parenthesis, open quotes, Hello, close quotes, right parenthesis, semicolon. Line 31, indented 3 times: if, left parenthesis, a e, dot, get Source, left parenthesis, right parenthesis, equals, equals, b 2, right parenthesis. Line 32, indented 4 times: System, dot, out, dot, print l n, left parenthesis, open quotes, Hello again, close quotes, right parenthesis, semicolon. Line 33, indented twice: right brace. Line 34, indented once: right brace. Line 35, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 36, indented once:

Left brace. Line 37, indented twice: Game, g, equals, new, Game, left parenthesis, right parenthesis, semicolon. Line 38, indented twice: g, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 39, indented once: right brace. Line 40: right brace.

Back to Code

Code to consider for questions 22 to 26. Program code. In the code, the words in the variable names are merged. Line 1: import, java x, dot, swing, dot, asterisk, semicolon. Line 2 import, java, dot, a w t, dot, asterisk, semicolon. Line 3: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 4: public, class, Game, extends, J Frame. Line 5: Left parenthesis. Line 6, indented once: private, J Check Box, c 1, comma, c 2, comma, c 3, semicolon. Line 7, indented once: private, i n t, value 1, comma, value 2, comma, value 3, semicolon. Line 8, indented once: private, Container, contents, semicolon. Line 9, indented once: public, Game, left parenthesis, right parenthesis. Line 10, indented once: Left brace. Line 11, indented twice: super, left parenthesis, open quotes, Play this game, close quotes, right parenthesis, semicolon. Line 12, indented twice: contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 13, indented twice: contents, dot, set Layout, left parenthesis, new, Flow Layout, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 14, indented twice: c 1, equals, new, J Check Box, left parenthesis, open quotes, Choice 1, close quotes, right parenthesis, semicolon. Line 15, indented twice: c 2, equals, new, J Check Box, left parenthesis, open quotes, Choice 2, close quotes, right parenthesis, semicolon. Line 16, indented twice: c 3, equals, new, J Check Box, left parenthesis, open quotes, Choice 3, close quotes, right parenthesis, semicolon. Line 17, indented twice: contents, dot, add, left parenthesis, c 1, right parenthesis, semicolon. Line 18, indented twice: contents, dot, add, left parenthesis, c 2, right parenthesis, semicolon. Line 19, indented twice: contents, dot, add, left parenthesis, c 3, right parenthesis, semicolon. Line 20, indented twice: My Handler, m h, equals, new, My Handler, left parenthesis, right parenthesis, semicolon. Line 21, indented twice: c 1, dot, add Item Listener, left parenthesis, m h, right parenthesis, semicolon. Line 22, indented twice: c 2, dot, add Item Listener, left parenthesis, m h, right parenthesis, semicolon. Line 23, indented twice: c 3, dot, add Item Listener, left parenthesis, m h, right parenthesis, semicolon. Line 24, indented twice: set Size, left parenthesis, 400, comma, 400, right parenthesis, semicolon. Line 25, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 26, indented once: right brace. Line 27, indented once: private, class, My Handler, implements, Item Listener. Line 28, indented once: Left brace. Line 29, indented twice: public, void, item State Changed, left parenthesis, Item Event, i e, right parenthesis. Line 30, indented twice: Left brace. Line 31, indented 3 times: if, left parenthesis, i e, dot, get Source, left parenthesis, right parenthesis, equals, equals, c 1, right parenthesis. Line 32, indented 3 times: Left brace. Line 33, indented 4 times: if, left parenthesis, i e, dot, get State Change, left parenthesis, right parenthesis, equals, equals, equals, equals, Item Event, dot, SELECTED, right parenthesis. Line 34, indented 5 times: value 1, equals, 1, semicolon. Line 35, indented 4 times: else. Line 36, indented 5 times: value 1, equals, 0, semicolon. Line 37, indented 3 times: right brace. Line 38, indented 3 times: else, if, left parenthesis, i e, dot, get Source, left parenthesis, right parenthesis, equals, equals, c 2, right parenthesis. Line 39, indented 3 times: Left brace. Line 40, indented 4 times: if, left parenthesis, i e, dot, get State Change, left parenthesis, right parenthesis, equals, equals, Item Event, dot, SELECTED, right parenthesis. Line 41, indented 5 times: value 2, equals, 2, semicolon. Line 42, indented 4 times: else. Line 43, indented 5 times: value 2, equals, 0, semicolon. Line 44, indented 3 times: right brace. Line 45, indented 3 times: else, if, left parenthesis, i e, dot, get Source, left parenthesis, right parenthesis, equals, equals, c 3, right parenthesis. Line 46, indented 3 times: left brace. Line 47, indented 4 times: if, left parenthesis, i e, dot, get State Change, left parenthesis, right parenthesis, equals, equals, Item Event, dot, SELECTED, right parenthesis. Line 48, indented 5 times: value 3, equals, 4, semicolon. Line 49, indented 4 times: else. Line 50, indented 5 times: value 3, equals, 0, semicolon. Line 51, indented 3 times: right brace. Line 52, indented 3 times:

System, dot, out, dot, print l n, left parenthesis, left parenthesis, value 1, plus, value 2, plus, value 3, right parenthesis, right parenthesis, semicolon. Line 53, indented twice: right brace. Line 54, indented once: right brace. Line 55, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 56, indented once: left brace. Line 57, indented twice: Game, g, equals, new, Game, left parenthesis, right parenthesis, semicolon. Line 58, indented twice: g, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 59, indented once: right brace. Line 60: right brace.

Back to Code

Code to consider for questions 27 to 32. Program code. In the code, the words in the variable names are merged. Line 1: import, java x, dot, swing, dot, asterisk, semicolon. Line 2: import, java, dot, a w t, dot, asterisk, semicolon. Line 3: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 4: public, class, A, extends, J Frame. Line 5: Left brace. Line 6, indented once: private, Container, c, semicolon. Line 7, indented once: private, J Button, b, semicolon. Line 8, indented once: private, J Text Field, t f, semicolon. Line 9: right brace.

Back to Code

Code for question 27. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 28. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 29. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 30. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 31. Program code. In the code, the words in the variable names are merged. Line 1: public, void, action Performed, left parenthesis, Action Event, a e, right parenthesis. Line 2: Left brace. Line 3, indented once: Forward slash, forward slash, your code goes here. Line 4: right brace.

Back to Code

Code for question 32. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, the My Handler class is a private class implementing. Line 2: Forward slash, forward slash, Action Listener. Line 3: My Handler, m h, equals, new, My Handler, left parenthesis, right parenthesis, semicolon. Line 4: Forward slash, forward slash, your code goes here.

Back to Code

Code to consider for questions 33 to 41. Program code. In the code, the words in the variable names are merged. Line 1: import, java x, dot, swing, dot, asterisk, semicolon. Line 2: import, java, dot, a w t, dot, asterisk, semicolon. Line 3: public, class, B, extends, J Frame. Line 4: Left brace. Line 5, indented once: private, Container, c, semicolon. Line 6, indented once: private, J Panel, p 1, semicolon. Line 7, indented once: private, J Panel, p 2, semicolon. Line 8, indented once: private, J Button, left square brace, right square brace, buttons, semicolon, forward slash, forward slash, length 12. Line 9, indented once: private, J Text Field, left square brace, right square brace, text fields, semicolon, forward slash, forward slash, length 10. Line 10, indented once: private, J Label, label 1, semicolon. Line 11, indented once: private, J Label, label 2, semicolon. Line 12: Right brace.

Back to Code

Code for question 33. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 34. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 35. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 36. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 37. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 38. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 39. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 40. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 41. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, your code goes here.

Back to Code

Code for question 42. Program code. In the code, the words in the variable names are merged. Line 1: Line 1: import, java, dot, swing, dot, asterisk, semicolon. Line 2: public, class, My Game, extends, J Frame. Line 3: Left brace. Line 4, indented once: Forward slash, forward slash, some code here. Line 5: Right brace.

Back to Code

Code for question 43. Program code. In the code, the words in the variable names are merged. Line 1: import, java x, dot, swing, dot, asterisk, semicolon. Line 2: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 3: public, class, My Game, extends, J Frame. Line 4: Left brace. Line 5, indented once: Forward slash, forward slash, some code here. Line 6, indented once: private, class, My Handler, extends, Action Listener. Line 7, indented once: Left brace. Line 8, indented twice: public, void, action Performed, left parenthesis, Action Event, a e, right parenthesis. Line 9, indented twice: left brace, right brace. Line 10, indented once: right brace. Line 11: right brace.

Back to Code

Code for question 44. Program code. In the code, the words in the variable names are merged. Line 1: import, java x, dot, swing, dot, asterisk, semicolon. Line 2: import, java, dot a w t, dot, event, dot, asterisk, semicolon. Line 3: public, class, My Game, extends, J Frame. Line 4: Left brace. Line 5, indented once: Forward slash, forward slash, some code here. Line 6, indented once: private, class, My Handler, implements, Item Listener. Line 7, indented once: Left brace. Line 8, indented twice: public, void, action Performed, left parenthesis, Action Event, a e, right parenthesis. Line 9, indented twice: Left brace, right brace. Line 10, indented once: right brace. Line 11: right brace.

Back to Code

Code for question 45. Program code. In the code, the words in the variable names are merged. Line 1: import, java x, dot, swing, dot, asterisk, semicolon. Line 2: import, java, dot a w t, dot, event, dot, asterisk, semicolon. Line 3: public, class, My Game, extends, J Frame. Line 4: Left brace. Line 5, indented once: Forward slash, forward slash, some code here. Line 6, indented once: private, class, My Handler, implements, Item Listener. Line 7, indented once: Left brace. Line 8, indented twice: public, void, item State Changed, left parenthesis, Action Event, a e, right parenthesis. Line 9, indented twice: Left brace, right brace. Line 10, indented once: right brace. Line 11: right brace.

Back to Code

Code for the program My Game dot java in question 46. Program code. In the code, the words in the variable names are merged. Line 1: import, java x, dot, swing, dot, asterisk, semicolon. Line 2: import, java, dot, a w t, dot, asterisk, semicolon. Line 3: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 4: public, class, My Game, extends, J Frame. Line 5: Left brace. Line 6, indented once: private, Container, c, semicolon. Line 7, indented once: private, J Label, l, semicolon. line 8, indented once: public, My Game, left parenthesis, right parenthesis. Line 9, indented once: Left brace. Line 10, indented twice: super, left parenthesis, open quotes, My Game, close quotes, right parenthesis, semicolon. Line 11, indented twice: c, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 12, indented twice, left parenthesis, right parenthesis, semicolon. Line 13, indented twice: c, dot, set Layout, left parenthesis, new, Flow Layout, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 14, indented twice: c, dot, add, left parenthesis, l, right parenthesis, semicolon, forward slash, forward slash, Line 15. Line 15, indented once: right brace. Line 16, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 17, indented once: Left brace. Line 18, indented twice: My Game, m g, equals, new, My Game, left parenthesis, right parenthesis, semicolon, forward slash, forward slash, Line 20. Line 19, indented once: right brace. Line 20: Left brace.

Back to Code

Run time error raised by the program in question 46, My Game dot java. Exception in thread "main" java dot lang dot Null Pointer Exception: Cannot read field "parent" because "comp" is null at java dot a w t dot Container dot add I m p l (Container dot Java:1121), at java dot a w t dot Container dot add (Container dot Java:440), at My Game. init (My Game dot java:15), at My Game dot main (My Game dot java:20).

Back to Code

Code in question 47. Program code. In the code, the words in the variable names are merged. Line 1: import java x, dot, swing, dot, asterisk, semicolon. Line 2: public, class, My Game, extends, J Frame. Line 3: Left brace. Line 4, indented once: public, My Game, left parenthesis, right parenthesis. Line 5, indented once: Left brace. Line 6, indented twice: super, left parenthesis, open quotes, My Game, close quotes, right parenthesis, semicolon. Line 7, indented twice: set Size, left parenthesis, 400, comma, 400, right parenthesis, semicolon. Line 8, indented once: right brace. Line 9, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 10, indented once: left brace. Line 11, indented twice: My Game, m g, equals, new, My Game, left parenthesis, right parenthesis, semicolon. Line 12, indented once: right brace. Line 13: right brace.

Back to Code

Code for the program My Game dot java in question 48. Program code. In the code, the words in the variable names are merged. Line 1: import, java x, dot, swing, dot, asterisk, semicolon. Line 2: import, java, dot, a w t, dot, asterisk, semicolon. Line 3: public, class, My Game, extends, J Frame. Line 4: Left brace. Line 5, indented once: private, Container, c, semicolon. Line 6, indented once: private, J Text Field, t f, semicolon. Line 7, indented once: public, My Game, left parenthesis, right parenthesis. Line 8, indented once: left brace. Line 9, indented twice: super, left parenthesis, open quotes, My

Game, close quotes, right parenthesis, semicolon. Line 10, indented twice: c, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 11, indented twice: t f, equals, new, J Text Field, left parenthesis, open quotes, Hello, close quotes, right parenthesis, semicolon. Line 12, indented twice: c, dot, add, left parenthesis, t f, comma, NORTH, right parenthesis, semicolon. Line 13, indented twice: set Size, left parenthesis, 400, comma, 400, right parenthesis, semicolon. Line 14, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 15, indented once: Left brace. Line 16, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 17, indented once: Left brace. Line 18, indented twice: My Game, m g, equals, new, My Game, left parenthesis, right parenthesis, semicolon. Line 19, indented once: right brace. Line 20: Left brace.

Back to Code

The error that is generated when the code in question 48 is compiled. The error indicates that the the variable NORTH in line number 14 of the program My Game dot java cannot be found. Program code. In the code, the words in the variable names are merged. c, dot, add, left parenthesis, t f, comma, NORTH, right parenthesis.

Back to Code

Code in the program My Game dot java. Program code. In the code, the words in the variable names are merged. Line 1: import, java x, dot, swing, dot, asterisk, semicolon. Line 2: import, java, dot, a w t, dot, asterisk, semicolon. Line 3: public, class, My Game, extends, J Frame. Line 4: Left brace. Line 5, indented once: Forward slash, forward slash, Some code here. Line 6, indented once: private, class, My Handler, implements, Action Listener. Line 7, indented once: Left brace. Line 8, indented twice: public, void, action Performed, left parenthesis, Action Event, a e, right parenthesis. Line 9, indented twice: Left brace, right brace. Line 10, indented once: right brace. Line 11: right brace.

Back to Code

The errors that are generated when the code in question 49 is compiled. The first error indicates an error in the keyword Action Listener in line 7 of the program My Game dot java. Line 7 is listed further. Program code. In the code, the words in the variable names are merged. private, class, My Handler, implement, Action Listener. The second error indicates an error in the keywork Action Event in the following line of code in the same program. Program code. In the code, the words in the variable names are merged. public, void, action Preformed, left parenthesis, Action Event, right parenthesis, a e, right parenthesis.

Back to Code

Code for the program My Game dot java in question 50. Program code. In the code, the words in the variable names are merged. Line 1: import, java x, dot, swing, dot, asterisk, semicolon. Line 2: import, java, dot, a w t, dot, asterisk, semicolon. Line 3: import, java, dot, a w t, dot, event, dot, asterisk, semicolon. Line 4: public, class, My Game, extends, J Frame. Line 5: Left brace. Line 6, indented once: private, J Button, b, semicolon. Line 7, indented once: private, J Text Field, t f, semicolon. Line 8, indented once: private, Container, contents, semicolon. Line 9, indented once: public, My Game, left parenthesis, right parenthesis. Line 10, indented once: Left brace. Line 11, indented twice: super, left parenthesis, open quotes, Play this game, close quotes, right parenthesis, semicolon. Line 12, indented twice: contents, equals, get Content Pane, left parenthesis, right parenthesis, semicolon. Line 13, indented twice: contents, dot, set Layout, left parenthesis, new, Grid Layout, left parenthesis, 1, comma, 2, right parenthesis, right parenthesis, semicolon. Line 14, indented twice: b, equals, new, J Button, left parenthesis, open quotes, Click here, close quotes, right parenthesis, semicolon. Line 15, indented twice: t f, equals, new, J Text Field, left parenthesis, open quotes, Hello, close quotes, right parenthesis, semicolon. Line 16, indented twice: contents, dot, add, left parenthesis, b, right parenthesis, semicolon. Line 17, indented twice: contents, dot, add, left parenthesis, t f, right parenthesis, semicolon. Line 18, indented twice: set Size, left parenthesis, 400, comma, 400, right parenthesis, semicolon. Line 19, indented twice: set Visible, left parenthesis, true, right parenthesis, semicolon. Line 20, indented once: right brace. Line 21, indented once: private, class, My Handler implements, Action Listener. Line 22, indented once: Left brace. Line 23, indented twice: public, void, action Performed, left parenthesis, Action Event, a e, right parenthesis. Line 24, indented twice: Left brace. Line 25, indented 3 times: t f, dot, set Text, left parenthesis, open quotes, Hi, close quotes, right parenthesis, semicolon. Line 26, indented twice: right brace. Line 27, indented once: right brace. Line 28, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 29, indented once: Left brace. Line 30, indented twice: My Game, g, equals, new, My Game, left parenthesis, right parenthesis, semicolon. Line 31, indented once: g, dot, set Default Close Operation, left parenthesis, J Frame, dot, EXIT underscore ON underscore CLOSE, right parenthesis, semicolon. Line 32, indented once: right brace. Line 33: right brace.

Back to Code

The positions on a football field displayed as a button. The following positions are listed. Left wing of 11. Striker of 9. Right wing of 7. Left midfielder of 6. Midfielder of 10. Right midfielder of 8. Left defender of 3. Stopper of 4. Sweeper of 5. Right defender of 2. Goalie of 1.

Back to Figure

attribute Name, equals, double quotes, value, double quotes.

Back to Code

attribute Name, equals, single quote, value, single quote.

Back to Code

x m l n s, colon, f x, equals, double quotes, h t t p, colon, 2 forward slashes, java f x dot com, forward slash, f x m l, double quotes.

Back to Code

void calculate, left parentheses, Action Event event, right parentheses.

Back to Code

void changed, left parentheses, Observable Value, left angled bracket, question mark, extends uppercase t, right angled bracket, observable, comma, uppercase t old Value, comma, uppercase t, new Value, right parentheses.

Back to Code

Double quotes, style Attribute 1, colon, value 1, semicolon, style Attribute 2, colon, value 2, semicolon, style Attribute 3, colon, value 3, semicolon, dot, dot, dot, double quotes.

Back to Code

Double quotes, hyphen, f x, hyphen, background, hyphen, color, colon, blue, semicolon, hyphen, f x, hyphen, font, hyphen, size, colon, 25, double quotes.

Back to Code

Program code. The code contains the following keywords: new, public, void, false. Line 1: t dot set On Finished, left parentheses, new Event Handler, left angled bracket, Action Event, right angled bracket, left parentheses, right parentheses. Line 2: left brace. Line 3, indented once: at symbol, Override. Line 4, indented twice: public void handle, left parentheses, Action Event event, right parentheses. Line 5, indented twice: left brace. Line 6, indented thrice: disable Buttons, left parentheses, false, right parentheses, semicolon. Line 7, indented twice: right brace. Line 8, indented once: right brace. Line 9: right brace, semicolon.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: public, class, Controller. Line 2: Left brace. Line 3, indented once: private, Model Interface, model, semicolon, forward slash, forward slash, Model Interface is an interface. Line 4, indented once: private, View, view, semicolon. Line 5, indented once: dot,

dot, dot. Line 6: right brace.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: public, class, Model, implements, Model Interface. Line 2: Left brace. Line 3, indented once: Forward slash, forward slash, Implements Model Interface methods. Line 4, indented once: dot, dot, dot. Line 5: right brace.

Back to Code

The data in the initial two X Y Chart dot Series. Bonds series: “2020”,100.0 “2021”,103.0 “2022”,106.09 “2023”,109.2727. Stocks series: “2020”,100.0 “2021”,107.0 “2022”,114.49 “2023”,122.5043.

Back to Code

The general syntax to style a G U I component in a style sheet. Program code. In the code, the words in the variable names are merged. Line 1: selector. Line 2: left brace. Line 3, indented once: attribute 1, colon, value 1, semicolon. Line 4, indented once: attribute 2, colon, value 2, semicolon. Line 5, indented once: dot, dot, dot. Line 6: right brace.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: Line 1: Forward slash, forward slash, a few asterisks, Task 1: declare an H Box named top. Line 2: Forward slash, forward slash, also declare three Button instance variables. Line 3: Forward slash, forward slash, that will be added to the H Box top. Line 4: Forward slash, forward slash, These buttons will determine the grid size of the game: Line 5: Forward slash, forward slash, 3-by-3, 4-by-4, or 5-by-5. Line 6: Forward slash, forward slash, task 1 ends here.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, a few asterisks, Task 2: student code starts here. Line 2: Forward slash, forward slash, instantiate the Game View object. Line 3: Forward slash, forward slash, add game View to the center of this Border Pane. Line 4: Forward slash, forward slash, task 2 ends here.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, a few asterisks, Task 3: Student code restarts here. Line 2: Forward slash, forward slash, instantiate the H Box component named top. Line 3: Forward slash, forward slash, instantiate the Buttons that determine the grid size. Line 4: Forward slash, forward slash, add the buttons to H Box top. Line 5: Forward slash, forward slash, make them take all the available space. Line 6: Forward slash, forward slash, add H Box top to this Border Pane as its top component. Line 7: Forward slash, forward slash, task 3 ends here.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, a few asterisks, Task 4: Student code restarts here. Line 2: Forward slash, forward slash, create a private inner class that implements Event Handler. Line 3: Forward slash, forward slash, your method should identify which of the 3 buttons. Line 4: Forward slash, forward slash, was the source of the event. Line 5: Forward slash, forward slash, depending on which button was pressed, Line 6: Forward slash, forward slash, call the set Up Game method of the Game View class. Line 7: Forward slash, forward slash, with arguments 3, 4, or 5. Line 8: Forward slash, forward slash, the A P I of that method is: Line 9: Forward slash, forward slash, public, void, set Up Game, left parenthesis, i n t, n Sides, right parenthesis. Line 10: Forward slash, forward slash, task 4 ends here.

Back to Code

Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, a few asterisks, Task 5: Student code restarts here. Line 2: Forward slash, forward slash, Note: search for and complete Task 4 before performing this task. Line 3:

Forward slash, forward slash, declare and instantiate an Event Handler. Line 4: Forward slash, forward slash, register the handler on the 3 buttons. Line 5: Forward slash, forward slash, that you declared in Task 1. Line 6: Forward slash, forward slash, task 5 ends here.

Back to Code

Code in F X M L file to consider for questions 21 to 25. Program code. In the code, the words in the variable names are merged. Line 1: Left angle bracket, question mark, x m l, version, equals, open quotes, 1.0, close quotes, encoding, equals, open quotes, U T F hyphen 8, close quotes, question mark, right angle bracket. Line 2: left angle bracket, question mark, import, java f x, dot, scene, dot, control, dot, asterisk, question mark, right angle bracket. Line 3: Left angle bracket, question mark, import, java f x, dot, scene, dot, layout, dot, asterisk, question mark, right angle bracket. Line 4: Left angle bracket, H Box, f x, colon, controller, equals, open quotes, My Controller, close quotes. Line 5, indented a few times: x m l n s, colon, f x, equals, open quotes, h t t p, colon, double slash, java f x, dot, com, forward slash, f x m l, close quotes. Line 6, indented a few times: alignment, equals, open quotes, center, close quotes, spacing, equals, open quotes, 10, close quotes, right angle bracket. Line 7, indented once: left angle bracket, Button, f x, colon, i d, equals, open quotes, button 1, close quotes, text, equals, open quotes, Button 1, close quotes. Line 8, indented a few times: on Action, equals, open quotes, hash, go, close quotes, forward slash, right angle bracket. Line 9, indented once: left angle bracket, Button, f x, colon, i d, equals, open quotes, button 2, close quotes, text, equals, open quotes, Button 2, close quotes. Line 10, indented a few times: on Action, equals, open quotes, hash, go, close quotes, forward slash, right angle bracket. Line 11, indented once: Left angle bracket, Label, f x, colon, i d, equals, open quotes, result, close quotes, forward slash, right angle bracket. Line 12, indented once: Left angle bracket, Button, f x, colon, i d, equals, open quotes, button 3, close quotes, text, equals, open quotes, Button 3, close quotes. Line 13, indented a few times" on Action, equals, open quotes, hash, go, close quotes, forward slash, right angle bracket. Line 14, indented once: Left angle bracket, Button, f x, colon, i d, equals, open quotes, button 4, close quotes, text, equals, open quotes, Button 4, close quotes. Line 15, indented a few times: on Action, equals, open quotes, hash, go, close quotes, forward slash, right angle bracket. Line 16: Left angle bracket, forward slash, H Box, right angle bracket.

Back to Code

Code to consider for questions 26 to 29. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, Controller class. Line 2: Asterisk, Anderson, comma, Franceschi. Line 3: Asterisk, forward slash. Line 4: import, java f x, dot, event, dot, Action Event, semicolon. Line 5: import, java f x, dot, f x m l, dot, F X M L, semicolon. Line 6: import, java f x, dot, scene, dot, control, dot, asterisk, semicolon. Line 7: public, class, Controller. Line 8: Left brace. Line 9, indented once: at, F X M L, private, Check Box, c b 1, semicolon. Line 10, indented once: at, F X M L, private, Check Box, c b 2, semicolon. Line 11, indented once: at, F X M L, private, Check Box, c b 3, semicolon. Line 12, indented once: at, F X M L, private, Label, label, semicolon. Line 13, indented once: private, i n t, read, write, execute, semicolon. Line 14, indented once: at, F X M L, protected, void, mix, left parenthesis, Action Event, event, right parenthesis. Line 15, indented once: Left brace. Line 16, indented twice: Check Box, c b, equals, left parenthesis, Check Box, right parenthesis, event, dot, get Source, left parenthesis, right parenthesis, semicolon. Line 17, indented twice: if, left parenthesis, c b, equals, equals, c b 1, right parenthesis. Line 18, indented 3 times: read, equals, left parenthesis, c b, dot, is Selected, left parenthesis, right parenthesis, question mark, 4, colon, 0, right parenthesis, semicolon. Line 19, indented twice: else, if, left parenthesis, c b, equals, equals, c b 2, right parenthesis. Line 20, indented 3 times: write, equals, left parenthesis, c b, dot, is Selected, left parenthesis, right parenthesis, question mark, 2, colon, 0, right parenthesis, semicolon. Line 21, indented twice: else, if, left parenthesis, c b, equals, equals, c b 3, right parenthesis. Line 22, indented 3 times: execute, equals, left parenthesis, c b, dot, is Selected, left parenthesis, right parenthesis, question mark, 1, colon, 0, right parenthesis, semicolon. Line 23, indented twice: i n t, mode, equals, read, plus, write, plus, execute, semicolon. Line 24, indented 3 times: label, dot, set Text, left parenthesis, open quotes, mode, colon, space, close quotes, plus, mode, right parenthesis, semicolon. Line 25, indented once: right brace. Line 26: right brace.

Back to Code

Code to consider for questions 30 to 32. Program code. In the code, the words in the variable names are merged. Line 1: import, java f x, dot, event, dot, asterisk, semicolon. Line 2: import, java f x, dot, scene, dot, control, dot, Button, semicolon. Line 3: import, java f x, dot, scene, dot, layout, dot, asterisk, semicolon. Line 4: public, class, Board View, extends, Grid Pane. Line 5: Left brace. Line 6, indented once: private, Button, left square brace, right square brace, left square brace, right square brace, buttons, semicolon. Line 7, indented once: public, Board View, left parenthesis, right parenthesis. Line 8, indented once: left brace. Line 9, indented twice: super, left parenthesis, right parenthesis, semicolon. Line 10, indented twice: Column Constraints, c o l, equals, new, Column Constraints, left parenthesis, right parenthesis, semicolon. Line 11, indented twice: c o l, dot, set Percent Width, left parenthesis, 25, right parenthesis, semicolon. Line 12, indented twice: Row Constraints, row, equals, new, Row Constraints, left parenthesis, right parenthesis, semicolon. Line 13, indented twice: row, dot, set Percent Height, left parenthesis, 20, right parenthesis, semicolon. Line 14, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, 5, semicolon, i, plus, plus, right parenthesis. Line 15, indented 3 times: get Row Constraints, left parenthesis, right parenthesis, dot, add, left parenthesis, row, right parenthesis, semicolon. Line 16, indented twice: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, 4, semicolon, j, plus, plus, right parenthesis. Line 17, indented 3 times: get Column Constraints, left parenthesis, right parenthesis, dot, add, left parenthesis, c o l, right parenthesis, semicolon. Line 18, indented twice: buttons, equals, new, Button, left square brace, 5, right square brace, left square brace, 4, right square brace, semicolon. Line 19, indented twice: Button Handler, b h, equals, new, Button Handler, left parenthesis, right parenthesis, semicolon. Line 20, indented twice: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, 5, semicolon, i, plus, plus, right parenthesis. Line 21, indented twice: Left brace. Line 22, indented 3 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, 4, semicolon, j, plus, plus, right parenthesis. Line 23, indented 3 times: Left brace. Line 24, indented 4 times: forward slash, forward slash, instantiate the buttons. Line 25, indented 4 times: buttons, left square brace, i, right square brace, left square brace, j, right square brace, equals, new, Button, left parenthesis, right parenthesis, semicolon. Line 26, indented 4 times: buttons, left square brace, i, right square brace, left square brace, j, right square brace, dot, set Max Width, left parenthesis, Double, dot, MAX underscore VALUE, right parenthesis, semicolon. Line 27, indented 4 times: buttons, left square brace, i, right square brace, left square brace, j, right square brace, dot, set Max Height, left parenthesis, Double, dot, MAX underscore VALUE, right parenthesis, semicolon. Line 28, indented a few times: add, left parenthesis, buttons, left square brace, i, right square brace, left square brace, j, right square brace, comma, j, comma, i, right parenthesis, semicolon. Line 29, indented 4 times: buttons, left square brace, i, right square brace, left square brace, j, right square brace, dot, set On Action, left parenthesis, b h, right parenthesis, semicolon. Line 30, indented 3 times: right brace. Line 31, indented twice: right brace. Line 32, indented once: right brace. Line 33, indented once: private, class, Button Handler, implements, Event Handler, left angle bracket, Action Event, right angle bracket. Line 34, indented once: Left brace. Line 35, indented twice: public, void, handle, left parenthesis, Action Event, event, right parenthesis. Line 36, indented twice: left brace. Line 37, indented 3 times: for, left parenthesis, i n t, i, equals, 0, semicolon, i, less than, buttons, dot, length, semicolon, i, plus, plus, right parenthesis. Line 38, indented 4 times: for, left parenthesis, i n t, j, equals, 0, semicolon, j, less than, buttons, left square brace, i, right square brace, dot, length, semicolon, j, plus, plus, right parenthesis. Line 39, indented 5 times: if, left parenthesis, event, dot, get Source, left parenthesis, right parenthesis, equals, equals, buttons, left square brace, i, right square brace, left square brace, j, right square brace, right parenthesis. Line

40, indented 6 times: buttons, left square brace, i, right square brace, left square brace, j, right square brace, dot, set Text, left parenthesis, open quote, close quotes, plus, String, dot, value Of, left parenthesis, i, plus, j, right parenthesis, right parenthesis, semicolon. Line 41, indented twice: right brace. Line 42, indented once: right brace. Line 43: right brace.

Back to Code

Code to consider for questions 33 to 35. Program code. In the code, the words in the variable names are merged. Line 1: Left angle bracket, question mark, x m l, version, equals, open quotes, 1.0, close quotes, encoding, equals, open quotes, U T F hyphen 8, close quotes, question mark, right angle bracket. Line 2: Left angle bracket, question mark, import, java f x, dot, scene, dot, control, dot, asterisk, question mark, right angle bracket Line 3: Left angle bracket, question mark, import, java f x, dot, scene, dot, layout, dot, asterisk, question mark, right angle bracket. Line 4: Left angle bracket, Border Pane, f x, colon, controller, equals, open quotes, My Controller, close quotes. Line 5, indented a few times: x m l n s, colon, f x, equals, open quotes, h t t p, colon, forward slash, forward slash, java f x, dot, com, forward slash, f x m l, close quotes, right angle bracket. Line 6, indented once: Left angle bracket, top, right angle bracket. Line 7, indented twice: Left angle bracket, H Box, alignment, equals, open quotes, center, close quotes, right angle bracket. Line 8, indented 3 times: Left angle bracket, Button, f x, colon, i d, equals, open quotes, button 1, close quotes, text, equals, open quotes, INSERT, close quotes, forward slash, right angle bracket. Line 9, indented twice: Left angle bracket, forward slash, H Box, right angle bracket. Line 10, indented once: left angle bracket, forward slash, top, right angle bracket. Line 11, indented once: Left angle bracket, left, right angle bracket. Line 12, indented twice: Left angle bracket, H Box, alignment, equals, open quotes, center, close quotes, right angle bracket. Line 13, indented 3 times: left angle bracket, Button, f x, colon, i d, equals, open quotes, button 2, close quotes, text, equals, open quotes, UPDATE, close quotes, on Action, equals, open quotes, hash, go, close quotes, forward slash, right angle bracket. Line 14, indented twice: left angle bracket, forward slash, H Box, right angle bracket. Line 15, indented once: Left angle bracket, forward slash, left, right angle bracket. Line 16, indented once: Left angle bracket, exclamation mark, hyphen, hyphen, answer to questions 33 to 35 go here, hyphen, hyphen, right angle bracket. Line 17: Left angle bracket, forward slash, Border Pane, right angle bracket.

Back to Code

Code to consider for questions 36 to 40. Program code. In the code, the words in the variable names are merged. Line 1: import, java f x, dot, event, dot, asterisk, semicolon. Line 2: import, java f x, dot, scene, dot, control, dot, asterisk, semicolon. Line 3: import, java f x, dot, scene, dot, layout, dot, asterisk, semicolon. Line 4:

public, class, A, extends, H Box. Line 5: Left brace. Line 6, indented once: private, Button, b, semicolon. Line 7, indented once: private, Label, l, semicolon. Line 8: right brace.

Back to Code

Code in question 36. Program code. In the code, the words in the variable names are merged. Line 1: forward slash, forward slash, your code goes here.

Back to Code

Code in question 37. Program code. In the code, the words in the variable names are merged. Line 1: forward slash, forward slash, your code goes here.

Back to Code

Code in question 38. Program code. In the code, the words in the variable names are merged. Line 1: forward slash, forward slash, your code goes here.

Back to Code

Code in question 39. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, the My Handler class is a private class implementing Event Handler. Line 2: My Handler, m h, equals, new, My Handler, left parenthesis, right parenthesis, semicolon. Line 3: Forward slash, forward slash, your code goes here.

Back to Code

Code in question 40. Program code. In the code, the words in the variable names are merged. Line 1: public, void, handle, left parenthesis, Action Event, a e, right parenthesis. Line 2: Left brace. Line 3, indented once: forward slash, forward slash, your code goes here. Line 4: right brace.

Back to Code

Code to consider for questions 41 to 46. Program code. In the code, the words in the variable names are merged. Line 1: import, java f x, dot, scene, dot, control, dot, asterisk, semicolon. Line 2: import, java f x, dot, scene, dot, layout, dot, asterisk, semicolon. Line 3: public, class, B, extends, Border Pane. Line 4: Left brace. Line 5, indented once: private, V Box, left, semicolon. Line 6, indented once: private, H Box, top, semicolon. Line 7, indented once: private, Button, left square brace, right square brace, buttons, semicolon, forward slash, forward slash, length 4. Line 8, indented once: private, Text Field, left square brace, right square brace, text fields, semicolon, forward slash, forward slash, length 3. Line 9, indented once: private, Label, label 1, semicolon. Line 10, indented once: private, Label, label 2, semicolon. Line 11, right brace.

Back to Code

Code in question 41. Program code. In the code, the words in the variable names are merged. Line 1: forward slash, forward slash, your code goes here.

Back to Code

Code in question 42. Program code. In the code, the words in the variable names are merged. Line 1: forward slash, forward slash, your code goes here.

Back to Code

Code in question 43. Program code. In the code, the words in the variable names are merged. Line 1: forward slash, forward slash, your code goes here.

Back to Code

Code in question 44. Program code. In the code, the words in the variable names are merged. Line 1: forward slash, forward slash, your code goes here.

Back to Code

Code in question 45. Program code. In the code, the words in the variable names are merged. Line 1: forward slash, forward slash, your code goes here.

Back to Code

Code in question 46. Program code. In the code, the words in the variable names are merged. Line 1: forward slash, forward slash, your code goes here.

Back to Code

Code in question 47. Program code. In the code, the words in the variable names are merged. Line 1: quit, dot, set On Action, left parenthesis, new, Event Handler, left angle bracket, Action Event, right angle bracket, left parenthesis, right parenthesis. Line 2, indented once: Left brace. Line 3, indented twice: at, Override. Line 4, indented twice: public, void, handle, left parenthesis, Action Event, event, right parenthesis. Line 5, indented twice: left brace. Line 6, indented 3 times: System, dot, exit, left parenthesis, 0, right parenthesis, semicolon. Line 7, indented twice: right brace. Line 8, indented once: right brace. Line 9: right parenthesis, semicolon.

Back to Code

X M L code in question 52. Program code. In the code, the words in the variable names are merged. Line 1: Left angle bracket, question mark, x m l, version, equals, open quotes, 1.0, close quotes, encoding, equals, open quotes, U T F hyphen 8, close quotes, question mark, question mark, right angle bracket. Line 2: Left angle bracket, H Box, x m l n s, colon, f x, equals, open quotes, h t t p, colon, forward slash, forward slash, java f x, dot, com, forward slash, f x m l, close quotes. Line 3, indented twice: alignment, equals, open quotes, center, close quotes, spacing, equals, open quotes, 10, close quotes, right angle bracket. Line 4, indented once: left angle bracket, Button, text, equals, open quotes, Button 1, close quotes, forward slash, right angle bracket. Line 5: Left angle bracket, forward slash, H Box, right angle bracket.

Back to Code

X M L code in question 53. Program code. In the code, the words in the variable names are merged. Line 1: Left angle bracket, question mark, x m l, version, equals, open quotes, 1.0, close quotes, encoding, equals, open quotes, U T F hyphen 8, close quotes, question mark, right angle bracket. Line 2: Left angle bracket, question mark, import, java f x, dot, scene, dot, control, dot, asterisk, question mark, right angle bracket. Line 3: Left angle bracket, question mark, import, java f x, dot, scene, dot, layout, dot, asterisk, question mark, right angle bracket. Line 4: Left angle bracket, H Box, right angle bracket. Line 5, indented once: Left angle bracket, Label, text, equals, result, forward slash, right angle bracket. Line 6: Left angle bracket, forward slash, H Box, right angle bracket.

Back to Code

Program code in question 54. Program code. In the code, the words in the variable names are merged. Line 1: import, java, dot, scene, dot, layout, dot, asterisk, semicolon. Line 2: public, class, My Game, extends, Grid Pane. Line 3: Left brace. Line 4: Right brace.

Back to Code

Program code in question 55. Program code. In the code, the words in the variable names are merged. Line 1: import, java f x, dot, event, dot, asterisk, semicolon. Line 2: import, java f x, dot, scene, dot, control, dot, Button, semicolon. Line 3: import, java f x, dot, scene, dot, layout, dot, asterisk, semicolon. Line 4: public, class, My Game, extends, Grid Pane. Line 5: Left brace. Line 6, indented once: Forward slash, forward slash. some code here. Line 7, indented once: private, class, My Handler, extends, Event Handler, left angle bracket, Action Event, right angle bracket. Line 8, indented once: Left brace. Line 9, indented twice: public, void, handle, left parenthesis, Action Event, a e, right parenthesis. Line 10, indented twice: Left brace, right brace. Line 11, indented once: right brace. Line 12: right brace.

Back to Code

Program code in question 56. Program code. In the code, the words in the variable names are merged. Line 1: import, java f x, dot, event, dot, asterisk, semicolon. Line 2: import, java f x, dot, scene, dot, control, dot, Button, semicolon. Line 3: import, java f x, dot, scene, dot, layout, dot, asterisk, semicolon. Line 4: public, class, My Game, extends, Grid Pane. Line 5: Left brace. Line 6, indented once: Forward slash, forward slash. some code here. Line 7, indented once: private, class, My Handler, implements, Event Handler. Line 8, indented once: Left brace. Line 9, indented twice: public, void, handle, left parenthesis, Action Event, a e, right parenthesis. Line 10, indented twice: Left brace, right brace. Line 11, indented once: right

brace. Line 12: right brace.

Back to Code

Program code in question 57. Program code. In the code, the words in the variable names are merged. Line 1: X Y Chart, dot, Series, left angle bracket, String, comma, Number, right angle bracket, series. Line 2, indented once: equals, new, X Y Chart, dot, Series, left angle bracket, String, comma, Number, right angle bracket, left parenthesis, right parenthesis, semicolon. Line 3: series, dot, add, left parenthesis, new, X Y Chart, dot, Data, left angle bracket, String, comma, Number, right angle bracket, left parenthesis, open quotes, HI, close quotes, comma, 2.0, right parenthesis, right parenthesis, semicolon.

Back to Code

Program code in question 58. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, b c is a Bar Chart and has been instantiated. Line 2: Array List, left angle bracket, X Y Chart, dot, Series, left angle bracket, String, comma, Number, right angle bracket, right angle bracket, series. Line 3, indented once: equals, new, Array List, left angle bracket, X Y Chart, dot, Series, left angle bracket, String, comma, Number, right angle bracket, right angle bracket, right parenthesis, right parenthesis, semicolon. Line 4: Forward slash, forward slash, fill in series here. Line 5: b c, dot, add All, left parenthesis, series, right parenthesis, semicolon.

Back to Code

Program code in question 59. Program code. In the code, the words in the variable names are merged. Line 1: import, java, dot, net, dot, U R L, semicolon. Line 2: import, java f x, dot, application, dot, Application, semicolon. Line 3: import, java f x, dot, f x m l, dot, F X M L Loader, semicolon. Line 4: import, java f x, dot, scene, dot, layout, dot, V Box, semicolon. Line 5: import, java f x, dot, scene, dot, Scene, semicolon. Line 6: import, java f x, dot, stage, dot, Stage, semicolon. Line 7: public, class, Test 59, extends, Application. Line 8: Left brace. Line 9, indented once: at, Override. Line 10, indented once: public, void, start, left parenthesis, Stage, stage, right parenthesis. Line 11, indented once: left brace. Line 12, indented twice: try. Line 13, indented twice: Left brace. Line 14, indented 3 times: U R L, u r l, equals, get Class, left parenthesis, right parenthesis, dot, get Resource, left parenthesis, open quotes, f x m l underscore e x 1, dot, f x m l, close quotes, right parenthesis, semicolon. Line 15, indented 3 times: V Box, root, equals, F X M L Loader, dot, load, left parenthesis, u r l, right parenthesis, semicolon. Line 16, indented 3 times: Scene, scene, equals, new, Scene, left parenthesis, root, comma, 300, comma, 275, right parenthesis, semicolon. Line 17, indented 3 times: stage, dot, set Title, left parenthesis, open quotes, Test, close quotes, right parenthesis, semicolon. Line 18, indented 3 times: stage, dot, set Scene, left parenthesis, scene, right parenthesis, semicolon. Lien 19, indented 3 times: stage, dot, show, left parenthesis, right parenthesis, semicolon. Line 20, indented twice: right brace. Line 21, indented twice: catch, left parenthesis, Exception, e, right parenthesis. Line 22, indented twice: Left brace. Line 23, indented 3 times: System, dot, out, dot, print l n, left parenthesis, e, dot, get Message, left parenthesis, right parenthesis, right parenthesis, semicolon. Line 24, indented twice: right brace. Line 25, indented once: right brace. Line 26, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 27, indented once: left brace. Line 28, indented twice: launch, left parenthesis, a r g s, right parenthesis, semicolon. Line 29, indented once: right brace. Line 30: right brace.

Back to Code

Error message during compilation of code in question 59. Program code. In the code, the words in the variable names are merged. Line 1: java, dot, lang, dot, Null Pointer Exception, colon, Location is required.

Back to Code

X M L code in question 60. Program code. In the code, the words in the variable names are merged. Line 1: Left angle bracket, question mark, x m l, version, equals, open quotes, 1.0, close quotes, encoding, equals, open quotes, U T F hyphen 8, close quotes, question mark, right angle bracket. Line 2: Left angle bracket, question mark, import, java f x, dot, scene, dot, control, dot, asterisk, question mark, right angle bracket. Line 3: Left angle bracket, question mark, import, java f x, dot, scene, dot, layout, dot, asterisk, question mark, right angle bracket. Line 4: Left angle bracket, V Box, x m l n s, colon, f x, equals, open quotes, h t t p, colon, forward slash, forward slash, java f x, dot, com, forward slash, f x m l, close quotes. Line 5, indented a few times: alignment, equals, open quotes, center, close quotes, spacing, equals, open quotes, 20, close quotes, right angle bracket. Line 6, indented once: Label, text, equals, Welcome, forward slash, right angle bracket. Line 7, indented once: Left angle bracket, Label, text, equals, open quotes, F X M L Test, close quotes, forward slash, right angle bracket. Line 8: Left angle bracket, forward slash, V Box, right angle bracket.

Back to Code

Error message when the code in question 60 is executed. Program code. In the code, the words in the variable names are merged. Line 1: java f x, dot, f x m l, dot, Load Exception, colon. Line 2: f x m l, colon, e x 60, dot, f x m l, colon, 6.

Back to Code

Code in X M L file and program in question 61. Code in the X M L file. Program code. In the code, the words in the variable names are merged. Line 1: Left angle bracket, question mark, x m l, version, equals, open quotes, 1.0, close quotes, encoding, equals, open quotes, U T F hyphen 8, close quotes, question mark, right angle bracket. Line 2: Left angle bracket, question mark, import, java f x, dot, scene, dot, control, dot, asterisk, question mark, right angle bracket. Line 3: Left angle bracket, question mark, import, java f x, dot, scene, dot, layout, dot, asterisk, question mark, right angle bracket. Line 4: Left angle bracket, V Box, f x, colon, controller, equals, open quotes, Controller 61, close quotes. Line 5, indented a few times: x m l n s, colon, f x, equals, open quotes, h t t p, colon, forward slash, forward slash, java f x, dot, com, forward slash, f x m l, close quotes. Line 6, indented a few times: alignment, equals, open quotes, center, close quotes, spacing, equals, open quotes, 10, close quotes, right angle bracket. Line 7, indented once: Left angle bracket, Button, f x, colon, i d, equals, open quotes, button, close quotes, text, equals, open quotes, GO, close quotes. Line 8, indented a few times: on Action, equals, open quotes, hash, go, close quotes, forward slash, right angle bracket. Line 9: Left angle bracket, forward slash, V Box, right angle bracket. Code in the program file. Program code. In the code, the words in the variable names are merged. Line 1: import, java f x, dot, event, dot, Action Event, semicolon. Line 2: import, java f x, dot, f x m l, dot, F X M L, semicolon. Line 3: import, java f x, dot, scene, dot, control, dot, asterisk, semicolon. Line 4: public, class, Controller 61. Line 5: Left brace. Line 6, indented once: protected, void, go, left parenthesis, Action Event, event, right parenthesis. Line 7, indented once: Left brace. Line 8, indented twice: System, dot, out, dot, print l n, left parenthesis, open quotes, Inside go, close quotes, right parenthesis, semicolon. Line 9, indented once: right brace. Line 10: right brace.

Back to Code

Error message when the code in question 60 is executed. Program code. In the code, the words in the variable names are merged. Line 1: java f x, dot, f x m l, dot, Load Exception, colon, Error resolving on Action, equals, open single quotes, hash, go, close single quotes, comma. Line 2: either the event handler is not in the Namespace or there is. Line 3: an error in the script.

Back to Code

Program code in question 62. Program code. In the code, the words in the variable names are merged. Line 1: import, java f x, dot, application, dot, Application, semicolon. Line 2: import, java f x, dot, scene, dot, Scene, semicolon. Line 3: import, java f x, dot, stage, dot, Stage, semicolon. Line 4: public, class, Test 62, extends, Application. Line 5: Left brace. Line 6, indented once: public, void, start, left parenthesis, Stage, stage, right parenthesis. Line 7, indented once: Left brace. Line 8, indented twice: E x 62, root, equals, new, E x 62, left parenthesis, right parenthesis, semicolon. Line 9, indented twice: Scene, scene, equals, new, Scene, left parenthesis, root, comma, 300, comma, 275, right parenthesis, semicolon. Line 10, indented twice: stage, dot, set Title, left parenthesis, open quotes, Test, close quotes, right parenthesis, semicolon. Line 11, indented twice: stage, dot, set Scene, left parenthesis, scene, right parenthesis, semicolon. Line 12, indented once: right brace. Line 13, indented once: public, static, void, main, left parenthesis, String, left square brace, right square brace, a r g s, right parenthesis. Line 14, indented once: Left brace. Line 15, indented twice: launch, left parenthesis, a r g s, right parenthesis, semicolon. Line 16, indented once: right brace. Line 17: right brace.

Back to Code

Program code in question 63. Program code. In the code, the words in the variable names are merged. Line 1: import, java f x, dot, scene, dot, control, dot, asterisk, semicolon. Line 2: import, java f x, dot, scene, dot, layout, dot, asterisk, semicolon. Line 3: import, java f x, dot, event, dot, asterisk, semicolon. Line 4: public, class, E x 63, extends, V Box. Line 5: Left brace. Line 6, indented once: private, Button, button, semicolon. Line 7, indented once: private, Label, label, semicolon. Line 8, indented once: public, E x 63, left parenthesis, right parenthesis. Line 9, indented once: left brace. Line 10, indented twice: super, left parenthesis, right parenthesis, semicolon. Line

11, indented twice: label, equals, new, Label, left parenthesis, open quotes, HI, close quotes, right parenthesis, semicolon. Line 12, indented twice: button, equals, new, Button, left parenthesis, open quotes, GO, close quotes, right parenthesis, semicolon. Line 13, indented twice: get Children, left parenthesis, right parenthesis, dot, add, left parenthesis, label, right parenthesis, semicolon. Line 14, indented twice: get Children, left parenthesis, right parenthesis, dot, add, left parenthesis, button, right parenthesis, semicolon. Line 15, indented once: right brace. Line 16, indented once: private, class, Button Handler, implements, Event Handler, left angle bracket, Action Event, right angle bracket. Line 17, indented once: left brace. Line 18, indented twice: public, void, handle, left parenthesis, Action Event, a e, right parenthesis. Line 19, indented twice: left brace. Line 20, indented 3 times: label, dot, set Text, left parenthesis, open quotes, Hello, close quotes, right parenthesis, semicolon. Line 21, indented twice: right brace. Line 22, indented once: right brace. Line 23: right brace.

Back to Code

Unix permission command format. c h mod, x y z, file name.

Back to Code

Contents of a file that contains a list of items to be auctioned. Line 1: Olds mobile, comma, olds mobile dot g i f, comma, 100. Line 2: World Cup soccer ticket, comma, soccer Tickets, dot, g i f, comma, 50. Line 3: Trip for 2 to Rome, comma, trip, dot, g i f, comma, 100.

Back to Code

An example illustration indicating the use of color in program code. The color illustrates the syntax of the code as listed in an I D E tool.

Back to Example

An example illustration meant to introduce how figures are used to illustrate and summarize the concept at hand. In this example, a flowchart is listed.

Back to Figure

An example illustration meant to introduce how tables are used to illustrate and summarize the concept at hand. In this example, a flowchart is listed. A table that details access modifiers is listed in this example.

Back to Table

The Look it Up icon that is used to indicate that the student may need to perform some research to answer a question and write the required code.

Back to Figure

A table listing a few simple concepts in Python and Java. Python: Hash is used for single line comments. Program code. In the code, the words in the variable names are merged. Line 1: Hash, This is a comment. Java: Forward slash, forward slash, is used for single line comments. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, forward slash, This is a comment. Python: Triple open single quotes and triple close single quotes are used to delimit multiple line comments. Program code. In the code, the words in the variable names are merged. Line 1: Triple open single quotes, This is a comment that spans over several lines, Triple close single quotes. Java: Forward slash, asterisk, and, asterisk, forward slash, are used to delimit multiple line comments. Program code. In the code, the words in the variable names are merged. Line 1: Forward slash, asterisk, This is a comment that spans over several lines, asterisk, forward slash. Python: There is no need to end a statement with a semicolon. Data types are implicit. We do not need to declare a variable, we just use it. Program code. In the code, the words in the variable names are merged. Line 1: age, equals, 18. Line 2: g p a, equals, 3.7. Java: Every statement ends with a semicolon. Data types are explicit. A variable must be declared before using it. Program code. In the code, the words in the variable names are merged. Line 1: i n t, age, semicolon. Line 2: age, equals, 18, semicolon. Line 3: double, g p a, equals, 3.7, semicolon. Python: Forward slash, is used floating point division, forward slash, forward slash, is used for integer division. Program code. In the code, the words in the variable names are merged. Line 1: average, equals, 373, forward slash, 4. Line 2: hash, floating point division. Line 3: quarters, equals, 102, forward slash, forward slash, 25. Line 4: hash, integer division. Java: The type of division (floating point or integer) is based on the data types of the two operands. Program code. In the code, the words in the variable names are merged. double, average, equals, 373.0, forward slash, 4, semicolon. Line 2: forward slash, forward slash, floating point division. Line 3: i n t, quarters, equals, 102, forward slash, 25, semicolon. Line 4: forward slash, forward slash, integer division. Python: Single quotes and double quotes have the same meaning. Program code. In the code, the words in the variable names are merged. Line 1: grade, equals, open single quotes, A, close single quotes. Line 2: city, equals, open quotes, New York, close quotes. Java: Single quotes are used for characters, double quotes are used for Strings. Program code. In the code, the words in the variable names are merged. Line 1: char, grade, equals, open single quotes, A, close single quotes, semicolon. Line 2: String, city, equals, open quotes, New York, close quotes, semicolon. Python: There is no formal concept of a constant, but using uppercase for a variable is good practice to denote that we should not change its value. Program code. In the code, the words in the variable names are merged. Line 1: DAYS, equals, 365. Line 2: Hash, Please do not change the. Line 3: Hash, value of DAYS. Java: The final keyword is used to declare constants. The value of a constant cannot be changed. Program code. In the code, the words in the variable names are merged. Line 1: final, i n t, DAYS, equals, 365, semicolon. Line 2: Forward slash, forward slash, DAYS' value cannot be. Line 3: Forward slash, forward slash, changed. Python: Indentation counts. Parentheses around the condition are not needed, and a colon precedes the indented statements. Program code. In the code, the words in the variable names are merged. Line 1: if, age, right angle bracket, equals, 18, colon. Line 2, indented once: print, left parenthesis, open quotes, You can vote, close quotes, right parenthesis. Line 3, indented once: print, left parenthesis, close quotes, Congratulations, close quotes, right parenthesis. Java: Indentation does not count, but we use it for style and readability. Parentheses are required around conditions. If there is more than one statement attached to a selection or looping construct, we use curly braces for the block. Program code. In the code, the words in the variable names are merged. Line 1: if, left parenthesis, age, right angle bracket, equals, 18, right parenthesis. Line 2: Left brace. Line 3, indented once: System, dot, out, dot, print l n, left parenthesis, open quotes, You can vote, close quotes, right parenthesis, semicolon. Line 3, indented once: System, dot, out, dot, print l n, left parenthesis, open quotes, Congratulations, close quotes, right parenthesis, semicolon. Line 4: right brace. Python: The logical operators are and, or, and not. Program code. In the code, the words in the variable names are merged.

Line 1: if, age, greater than or equals, 18, and, age, less than or equals, 65, colon. Line 2, indented once: print, left parenthesis, open quotes, Hi, close quotes, right parenthesis. Java: The logical operators are ampersand, ampersand (for and), pipe, pipe, (for or), and exclamation mark (for not). Program code. In the code, the words in the variable names are merged. Line 1: if, left parenthesis, age, greater than or equals, 18, and, age, less than or equals, 65, right parenthesis. Line 2, indented once: System, dot, out, dot, print l n, left parenthesis, open quotes, Hi, close quotes, right parenthesis, semicolon. Python: The keyword def is used for functions. The parameter list is a comma delimited list of variable names. The function header is followed by a colon. The body of the function is indented. Program code. In the code, the words in the variable names are merged. Line 1: def, sum, left parenthesis, a, comma, b, right parenthesis, colon. Line 2, indented once: return, a, plus, b. Java: The parameter list of a function (called method in Java) is a comma delimited list of variable names and their respective data types. The method header includes the return data type (void if the method does not return anything). The body of the method is inside a set of curly braces. Indentation is optional but recommended for better readability. Program code. In the code, the words in the variable names are merged. Line 1: i n t, sum, left parenthesis, i n t, a, comma, i n t, b, right parenthesis. Line 2: Left brace. Line 3, indented once: return, a, plus, b, semicolon. Line 4: right brace.

Back to Figure